International Journal of Advance Research in Science and Engineering
Volume No.07, Special Issue No.04, March 2018
www.ijarse.com
IJARSE
ISSN: 2319-8354

Is It Safe to Uplift a Patch? An Empirical Study on Trac

Mir Mohammad Yousuf [1],Dr. Satwinder Singh[2]
Department of Computer Science and Technology
School of Engineering and Technology,Central University of Punjab,Bathinda-India

ABSTRACT

The patches that fix the critical issues, or implement new functionality with high value features are frequently promoted directly from the development channel to a stabilization channel. This practice that takes place in the rapid development process, potentially by skipping one or more stabilization channels is called patch uplift. Patch uplift is mandatory to fix the bugs reported by the quality assurance team or by the users but at the same time is risky, because the patches that are rushed through the stabilization phase can end up introducing regressions in the code. This paper examines the patch uplift of the python software projects and tries to analyze whether the patch uplift increases or decreases the maintenance of the project. The result shows that as the patch uplift takes place, the main focus of the development team lies in improving the coding standards i.e., with the increase in patch uplifts, the code smells approximately reduce. The results also show the relationship between the coding standards followed, the new modules added and the number of the bugs reported.

Keywords: *Bugs, code smells, patch, python software project, quality assurance, regression.*

1. INTRODUCTION

As the law of software engineering says, “No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle”. As a man-made artifact, software suffers from different type of errors, referred to as software bugs, which cause the software to produce an incorrect or unexpected result, or to behave in an unintended way and significantly threaten not only the reliability but also the security of computer systems. Therefore, a software bug can be defined as a problem or an issue causing a program to crash or produce an invalid input. Bugs are detected either during the software development life cycle by the testing team or by the customers or users post release of the software. Once a bug is discovered and reported, it needs to be fixed by the developers. In particular, for bugs that have direct and severe impact on customers, vendors usually make releasing timely patches the highest priority in order to minimize the amount of system down time.

Software that falls to fulfill the specified requirements needs to be fixed. This can be because of the following reasons:

i. The mistake that has been committed between the initial requirements and the final operation of the software system.
ii. The unrealistic time schedule for the software development
iii. Lack of the designing experience
iv. Lack of version control
v. Lack of coding practice experience

vi. Buggy third party tools

vii. Last minute changes in the requirements

viii. Poor software testing skills

Today, software bugs remain a continuous and expensive fixture of industrial and open source software development. Using the software configuration management (SCM) systems, the software projects carefully control their changes and capture the bug reports using the bug tracking software such as Bugzilla. Thus recording which change in the SCM fixes a specific bug in the change tracking system. The major problem with the bug fix data is that it sheds no light on when a bug has been injected into the code and who injected it.

Fixing the bugs results in addition of a software patch. A software patch is a piece of software that is designed to update a computer program or its supporting data, to fix or improve it. A patch is added frequently to fix the security vulnerabilities and other bugs named as bug fixes, and to improve the usability and the performance. In addition to this the patches are also added to introduce the new features which sometimes can be risky if the patches are poorly designed. Fixing one problem can sometimes introduce new problems due to the poorly designed patches. In special cases, the updates may knowingly break the functionality of the product.

In this paper we conduct a series of quantitative and qualitative analysis to understand the effects of uplifting a software patch. Overall we analyzed 32 versions of Trac. Trac is an open source web-based project management and bug tracking system that is adopted by various organizations for the use of bug tracking system for both free and open-source software and proprietary projects and products. Trac is developed in python development framework and is used by Internet Research task Force, JQuery UI, and WordPress.

2. LITERATURE REVIEW

The software bugs and their effect on the software quality and maintenance has gradually increased the interest of the researchers for which they were interested lately. Although there has been a less amount of research done in analyzing the python software system as compared to other systems of other platforms. Matteo Orrú et al (2015) [9] presented a dataset of metrics of python systems. They built the corpus, discussed the main issues in creating it, and provided its description and limitation. They also suggested the use of the data set for the empirical studies of the python systems that allows reproducibility of thus lowers the cost of the experiments. Zhifei Chen et al (2016). [2] introduced 11 smells and described the detection strategy. They also implemented the detection tool namely Pysmell that was used to identify the code smells in five real world python systems. Their results showed that Pysmell could detect 285code smell instances in total with the average precision of 97.7%, which revealed that large classes and large methods are more prevalent. .Wanwangying et. al. 2017)[3] conducted an empirical study on the cross-project correlated bugs i.e., a pair of casually related issues reported to different projects, focusing on two aspects. One aspect being how developers track the root causes across the projects and the second aspect how the downstream developers coordinate to deal with the upstream bugs. They manually identified and inspected 271 of the cross project bugs in a scientific python ecosystem and conducted an online survey with 116 respondents in the scientific python community. Their empirical results revealed the common practices of the developers when fixing the cross-project bugs and provide suggestions for the ecosystem-wide tool supports. Dag et al (2012)[4] Tried to investigate whether the metrics are consistent among themselves and to which extent they predict maintenance effort at the entire system level. However, their results showed that the metrics were mutually consistent. Only the system size and low cohesion were associated with the increased maintenance effort. However, they also concluded that apart from the size, surrogate maintainability measures might not reflect the future maintenance effort. In order to use the surrogates they need to be evaluated first in the context for which they will be used. Their results suggest that the local improvements should be accompanied by an evaluation at the system level. .Varuna et al (2015)[10] assessed the correlation of the bug indicators (DIT, CBO, LoC) with software bugs and developed the software prediction models using these bug indicators as model inputs. They also compared the relative effectiveness of these bug indicators towards prediction of bugs in Camel and Ant projects.

Mortiz et al (2014) [7] empirically explored the problems fixed through the modern code reviews (MCR) in open source software systems (OSS) and tried to increase the understanding the practical benefits that the MCR process produces on the code reviews. They manually classified 1,400 changes taking place in the reviewed code from the two OSS projects into a validated categorization scheme. The results showed that the types of due to the modern code review process in OSS were very similar to those in the academic systems and the industry from the literature. Their main contribution included the change classification i.e, a validated change classification for the java and C projects to better understand the actual effects of code reviews on 1400 changes done in reviews. The similarity ratio of maintainability-related to the functional problems being 75:25. They also revealed that 7-35% of the code review comments were discarded and 10-22% of the changes were not triggered by an explicit comment. In addition to this found that the bug-fixing tasks lead to fewer changes and the tasks with more altered files and a higher code churn have more changes. Their results reveal that 75% changes are related to the evaluability of the system. The more code churn or the higher number of touched files in a task, the more they expected the review on average.

Zhifei Chen et al (2017) [11]defined and detected the code smells in the python programs and explored the effects of the python smells on software maintainability. They introduced the 10 code smells and established a metric based detection method with three filtering strategies that they used to specify the metric thresholds (Statistics-Based-Strategy, Experience-Based-Strategy and the Tuning-Machine-Strategy).They performed a comparative study to investigate that how these three detection strategies perform in detecting the python smells and how these smells affect the maintainability with different detection strategies. They utilized the corpus of 106 popular projects on GitHub. Their results showed that metric-based performed well in detecting python smells and Tuning Machine detection achieved some different smell occurrences. In addition to this, the Long Parameter List and Long method are more prevalent than other smells. Their results also showed that there are several kinds of smells that are more significantly related to the changes or the faults in the python module. . Nicole Vavrova et al (2016)[8] developed a tool called the design defect detector, which parses a python module, creates a code model of it and reports on the presence of various design defects found there. They however compared the design defect density in python to the design defect density in java and found that the density measured in python (6.07 defects per 10,000 lines of code) was slightly lower than the density measured in java (8.37 defects per 10,000 lines of code).

Marco castallucioo et al (2017)[12] conducted a series of qualitative and quantitative analysis to understand the decision making the process of patch uplift at Mozilla and characteristics of uplifted patches that introduce regressions. They analyzed 33664 issue reports in 17 versions of Mozilla over a period. They examined the characteristics of the uplifted patches that introduced regressions in the code and found that they are more complex than clean uplifts.

3. METHODOLOGY AND APPROACH

The understanding of how a software project has grown, software evolution research influences the history of changes and bug reports .Thus by examining the history of the changes made to a software project, it's possible to better understand the patterns of bug introduction. In addition to this, make the developers aware of working on the risky i.e., the bug-prone sections of the software project. Since the source code matters, a change in one file may result in the change in multiple files of the software. Thus, there is a chance that fixing a single bug may result in introduction of new bugs in the software. . However, the bugs need to be fixed to avoid causing a failure or minimizing the possibility of introducing new bugs in the software. As a result increasing the reliability of the software system.

3.1 Data collection

The source code of a project is an essential component for any type of analysis. The characteristics of a source code determines the types of results we can infer for them. However, the bug list for such projects can be essential component for ensuring better quality assurance of the product. If we select a project, big enough to represent an industry sized software, the inferences can hold true for the industry software as well as the academicians. Therefore, a person needs to be wise while selecting an open source project .We selected an open source project with some stability and proper documentation. We collected, from the Trac issue tracking system, all the bugs with the priority being highest, high, normal, low and lowest AND type of bug being defect, enhancement of 32 versions of the Trac. Table 1 shows the release dates of different versions of Trac that were analyzed.

S.NO	TRAC Version	Release Date
TRAC 0.0.x releases		
1.	0.12 'Babel'	(June 13, 2010)
2.	0.12.1	(October 9, 2010)
3.	0.12.2	(January 31, 2011)
4.	0.12.3	(February 6, 2012)

5.	0.12.4	(September 7, 2012)
6.	0.12.5	(January 15, 2013)
7.	0.12.6	(October 23, 2014)
8.	0.12.7	(July 12, 2015)
Trac 1.0.xx releases		
9.	1.0 "Cell"	(September 7, 2012)
10.	1.0.1	(February 1, 2013)
11.	1.0.2	(October 23, 2014)
12.	1.0.3	(January 13, 2015)
13.	1.0.4	(February 8, 2015)
14.	1.0.5	(March 24, 2015)
15.	1.0.6	(May 20, 2015)
16.	1.0.7	(July 17, 2015)
17.	1.0.8	(July 24, 2015)
18.	1.0.9	(September 10, 2015)
19.	1.0.10	(February 20, 2016)
20.	1.0.11	(May 7, 2016)
21.	1.0.12	(July 4, 2016)
22.	1.0.13	(September 11, 2016)
23.	1.0.14	(June 9, 2017)
24.	1.0.15	(June 16, 2017)
TRAC Versions1.1.x releases		

25.	1.1.1	(February 3, 2013)
26.	1.1.2	(October 23, 2014)
27.	1.1.3	(January 13, 2015)
28.	1.1.4	(March 24, 2015)
29.	1.1.5	(May 18, 2015)
30.	1.1.6	*(July 17, 2015)*
TRAC Versions1.2.x releases		
31.	1.2 "Hermes"	(November 5, 2016)
32.	1.2.1	(March 29, 2017)

Table 1: The release dates of different versions of Trac

3.2 Data extraction

The degree of the impact that a bug has on the development or operation of a component system is called the bug severity. Higher effect on the severity will lead to assignment of the higher to the bug. The quality assurance engineer usually determines the severity level of the bug. The bug severity classification in several types, the actual terminologies and their meaning can vary depending upon the people, projects, organizations, or defect tracking tools. The bugs of the above-mentioned versions were extracted depending upon the severity, type, priority of the bugs. The bugs are divided into two categories:

i. Release blocker:

Any software can have the bugs. All the bugs cannot be fixed otherwise the product would never be released. Those bugs that could block the release of the product is called the release blocker bug and has the following symptoms:

a) the application cannot be installed
b) the application does not start
c) the data is lost
d) the running of the application results in a crash
e) freezing of the application
f) the security bug
g) the license problem

The release blockers must fulfill the following conditions:

a) the problem must be reproducible by more users; if it is not, its most likely the problem on the other user side.

b) Broken functionality must be regression against the last released version; if it was not reported early, it is a rarely used function and the fix could not wait until the next release.

c) The problem must affect most users or there must not be a reasonable workaround; its bad to block the release for all users because of a corner case when there exists a reasonable workaround

ii. Normal Bugs:

Normal bugs are those that are not critical, major or release blocker. They have isolated impact and may have workarounds. Most of the issues are considered normal.

The normally accepted classification of the severity of the bugs is following:

a) Critical: If the bug affects the critical functionality or the critical data it's a critical bug. It can also lead to the complete shutdown of the process and nothing can proceed further. It does not have a work around. The examples are unsuccessful installation, complete failure of a future.

b) Major: If the bug affects the major functionality or the major data, it's a major bug. It has a highly sever defect and can collapse the system. However, certain parts of the system remain functional. For example a feature is not functional from one module but the task is doable if 10 complicated indirect steps are followed in another module/s.

c) Minor: If a bug affects a minor functionality or non-critical data, it is a minor bug. It causes some undesirable behavior but the system is still functional i.e., it has an easy work around. For example a minor functionality is function from one module but the same task is easily doable from another module

d) Trivial: It doesn't affect the functionality or the data. It does not impact productivity or efficiency. For example the petty layout disappearances, spelling/grammatical errors.

Bug priority can be defined as the order in which a bug should be fixed. Higher the priority sooner the bug will be resolved. Bug priority can be classified into three classes.

i. Highest**:** This bug is given the more priority to be fixed as soon as possible as it can affect the major functionality of the product.

ii. High: The Bug should be resolved as soon as possible as it affects the system severely and cannot be used until it's fixed.

iii. Normal**.** : During the normal course of the development activities, the bug should be fixed. It can wait until a new version is created.

iv. Low: The bug is irritant but can be fixed after the more serious are fixed.

v. Lowest**:** These types of bugs are given the least priority and are fixed after bugs with the low priority are fixed.

Apart from these priorities, one more class of bugs exist called the feature request/new feature/ enhancement bugs, where the users/testers request for the new features e.g., providing a better UX for creating, editing and managing draft revisions.

3.3 Bug association

The versions of the Django were selected on the basis if the substantial difference in the release dates and were examined for the presence of bugs. The bugs were selected depending upon the type and severity of the bug.

Version:0.12Babel',0.12.1,0.12.2,0.12.3.0,12.4.0,12.5,0.12.6,0.12.7,1.0,1.0.1,1.0.2,1.0.3,1.0.4,1.0.5,1.0.6,1.0.7,1.0.8,1.0.9,1.0.10,1.0.11,1.0.12,1.0.13,1.0.14,1.1.1,1.1.2,1.1.3,1.1.4,1.1.5,1.1.6,1.2 and 1.2.1.

Priority: Highest, High, Normal, Low, and Lowest.

Type: normal, enhancement and task.

Status: closed, assigned and new.

Resolution: fixed, invalid, won fix, duplicate, works fine and needs info.

After the mapping of the bugs was done with the parameters mentioned above, the resultant files contained the distribution of the defects per version as:

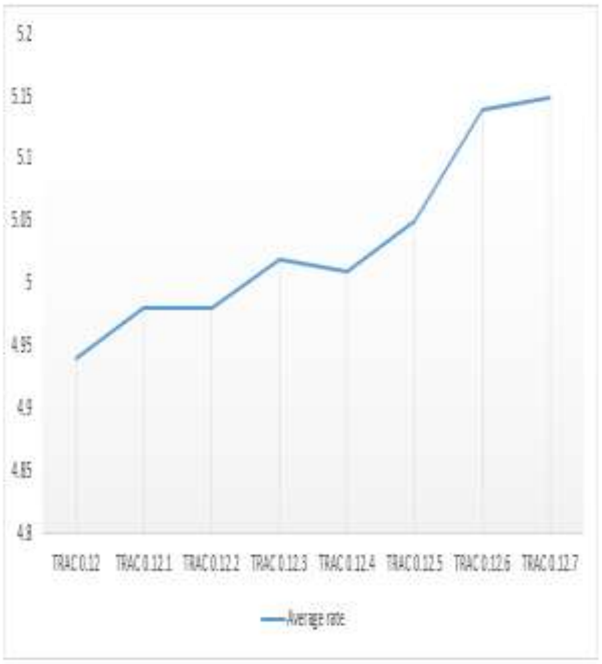
S.NO	TRAC version	Defects	Enhancements
1	TRAC 0.12.x	388	65
2	TRAC 1.0.x	292	42
3	TRAC 1.1.x	40	14
4	TRAC 1.2.x	23	26
TOTAL		743	123
		866	

Table2: The total number of bugs in different Trac versions

4. RESULTS AND DISCUSSION

The 32 versions of the Trac were tested on the pylint. A script was written in python that analyzes the source code of the entire python project looking for the signs of poor quality. Pylint is a source code, bug and the quality checker for the python programming language that follows the style recommended by PEP8 [13] (the official style guide of python core). The script tests each and every file and rates the code of the entire project depending upon the standards like if the variable names are well formed according to the project's coding standards or if declared interfaces are truly implemented [14] or not, etc. followed in the code. Table 2 shows the average rate of the 32 versions of Trac.

Figures 1-4 show the average rate of the code of the respective versions of Trac tested on the pylint. The elevating slope shows that the patch uplift results in improving the standards of the code i.e., it depicts with increase in the version of the project there are approximately lesser chances of the security vulnerabilities and bugs. It may be noted that the rate of none of the versions of the Trac has reached 7.0 indicating the fact that the software can never be bug free. The figures 5-8 show the relationship between the number of the module files and the number of the bugs reported. The graphs show that with increase in the patch uplift, the number of bugs are also reduced. The stabilization phase only focuses on the reliability of the product and tries to make the product bug free. Thus adding the patches till the stability is achieved.



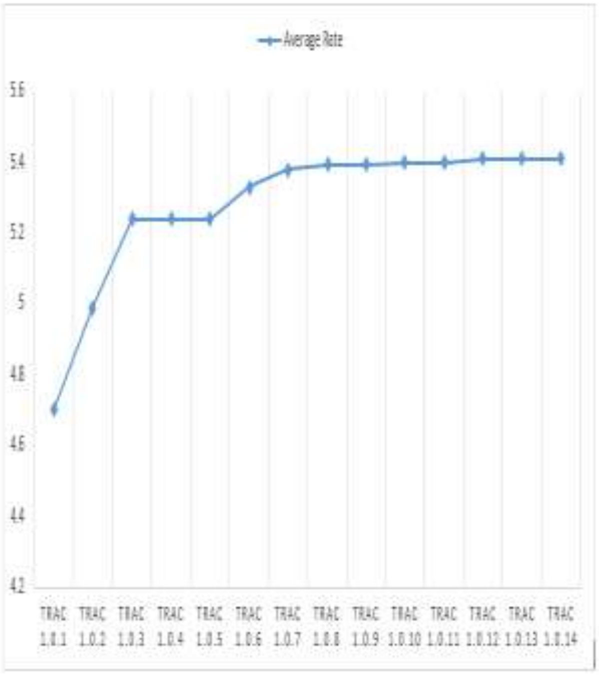
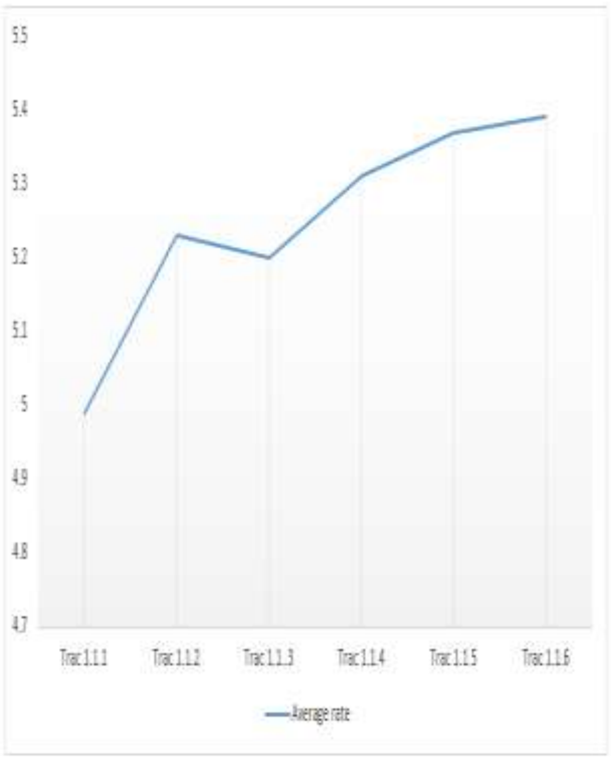


Figure 1: Average rate of Trac 0.12.xx versionsFigure 2: Average rate of Trac 1.0.xx versions



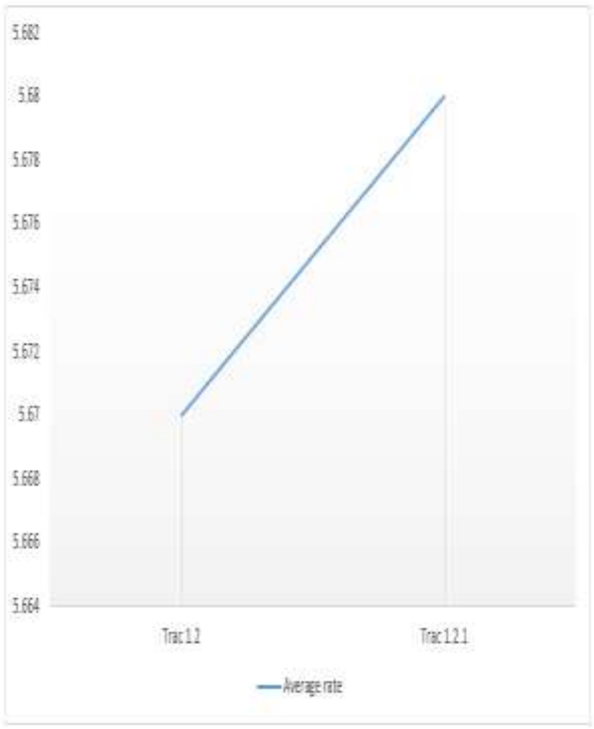
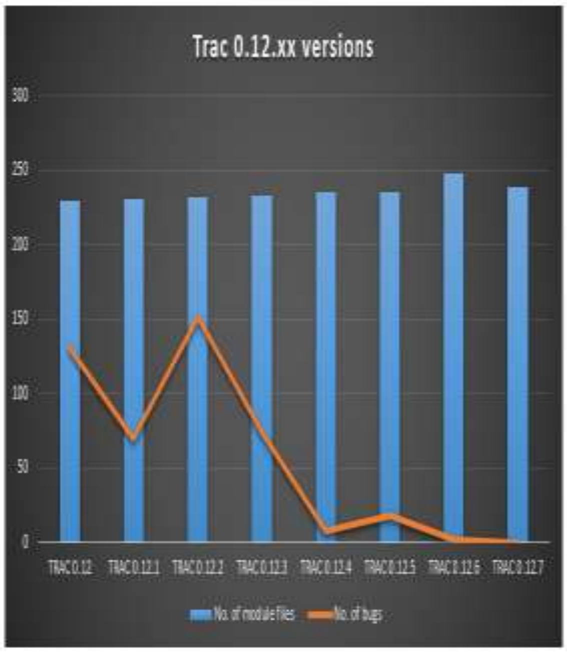


Figure 3: Average rate of Trac 1.1.xx versionsFigure 4: Average rate of Trac 1.1.xx versions



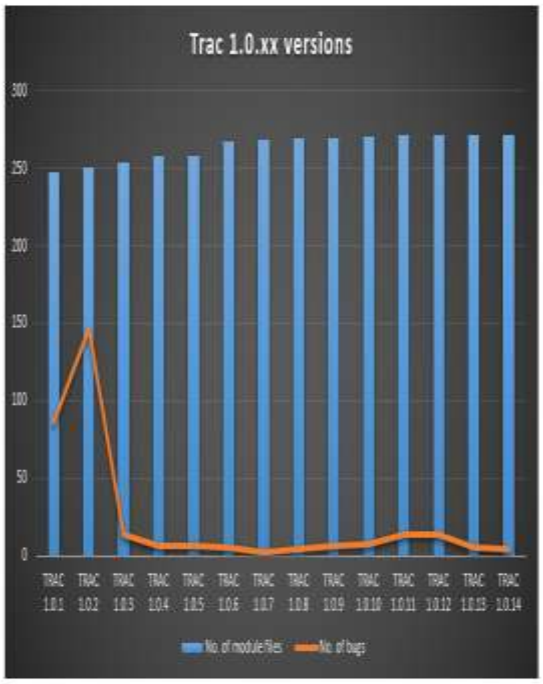
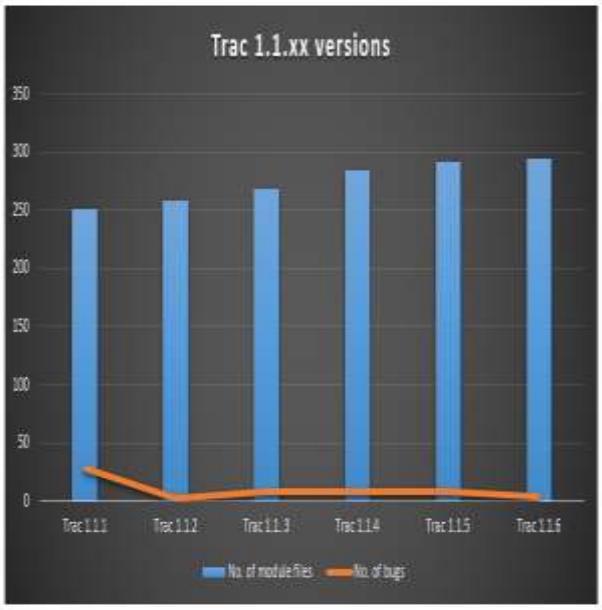


Figure 5. Number of modules vs number of bugs Figure 6. Number of modules vs number of reported in Trac 0.12.xx versionsbugs reported in Trac 1.0.xx versions



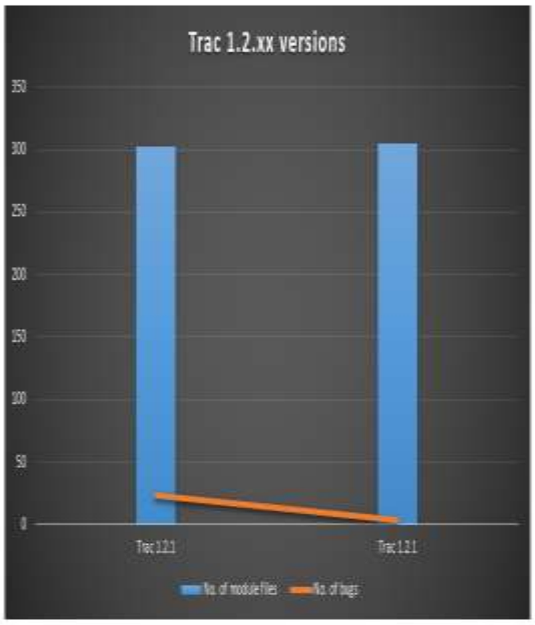


Figure 7. Number of modules vs number of bugs Figure 8. Number of modules vs number of bugs reported in Trac 1.1.xx versionsreported in Trac 1.2.xx versions

5. CONCLUSION AND FUTURE WORK

It is a well-known fact that the software maintenance is costly and effort intensive. Therefore, a software system should be maintainable. A lot of research has been done in the object-oriented systems written in different languages while less in software systems written in python. This paper analyzes the change in the bugginess of the python software system i.e., Trac. The results show that the patch uplifts not only increase the coding standards and the number of moSdules but fixes the bugs as well achieve the better quality of the software. A huge number of projects are developed using the python language paving the way to the researchers to analyze the effectiveness of these software. Not only the error-prone modules are troublesome, but there may be other factors as well that degrade the performance of the software. This work can be extended by evaluating the error prone modules in the software, analyzing which priority bugs involve more change in the lines of code and finding the correlation between the change in the number of lines of code due to fixing a bug and the number of bugs reported.

REFERENCES

[1] Abbes, M., Khomh, F., Gueheneuc, Y. G., & Antoniol, G. (2011). An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension. *15th European Conference on Software Maintenance and Reengineering (CSMR)* (pp. 181-190). Oldenburg: IEEE.

[2] Chen, Zhifei & Chen, Lin & Ma, Wanwangying & Xu, Baowen. (2016). Detecting Code Smells in Python Programs. 18-23. 10.1109/SATE.2016.10.

[3] Chen, Zhifei & Chen, Lin & Ma, Wanwangying & Xu, Baowen. (23-09-2017)Understanding metric-based detectable smellsin Python software: a comparative study.*In Information and Software Technology.*

[4] Dag I.K. Sjøberg,Bente Anda,Audris & Mockus.(2012)Questioning Software Maintenance Metrics:A Comparative Case Study.In *"Empirical Software Engineering and Measurement (ESEM), 2012 ACM-IEEE International Symposium on*.

[5] IEEE Std. 1219 (1993).: Standard for Software Maintenance. *IEEE Computer Society Pres.*,

[6] Marco,Le & Foutse (26 september 2017)“Is it safe to uplit this patch”arXiv:1709.00852v1[cs.SE] 26 Sep2017

[7] Moritz Beller, Alberto Bacchelli, Andy Zaidman & Elmar Juergens (2014).Modern code reviews in open-source projects: which problems do they fix?.*Proceedings of the 11th Working Conference on Mining Software Repositories*

[8] Nicole,Vadim (2016).Does python smell like java. In:*The art of Science and Engineering of programming.Vol I*,no.2,2017 article II;29 pages

[9] Orrù, Matteo & Tempero, Ewan & Marchesi, Michele & Tonelli, Roberto & Destefanis, Giuseppe. (2015). A Curated Benchmark Collection of Python Systems for Empirical Studies on Software Engineering.10.1145/2810146.2810148.

[10] Varuna ,Ganeshan ,Singhal "Developing Software Bug Prediction Models Using Various Software Metrics as the Bug Indicator"s.(IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 6, No. 2, 2015.

[11] W. Ma, L. Chen, X. Zhang, Y. Zhou & B. Xu, "How Do Developers Fix Cross-Project Correlated Bugs? A Case Study on the GitHub Scientific Python Ecosystem," *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Buenos Aires, 2017,pp.381-392.doi: 10.1109/ICSE.2017.42

[12] Castelluccio, Marco & An, Le & Khomh, Foutse. (2017). Is it Safe to Uplift this Patch?: An Empirical Study on Mozilla Firefox. 411-421. 10.1109/ICSME.2017.82.

[13] Pylint (analyzes Python source code looking for bugs and signs of poor quality)". Logilab.org. 2006-09-26. Retrieved 2018-03-01.

[14] Pylint User Manual – Pylint 2.0.0 documentation". Docs.pylint.org. Retrieved 2018-03-01.

[15] W. Li and S. Henry, "Maintenance metrics for the object oriented paradigm," in First International Software Metrics Symposium, 1993.

[16] M. Catwright and M. Shepperd, "An empirical investigation of an object oriented software system," IEEE Transactions on software engineering, vol. 26, no. 8, pp. 786-796, 2000.

[17] K. Dhambri, H. Sahraoui and P. Poulin, "Visual detection of design anomalies," in 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), 2008.