An empirical study on the impact of Python dynamic features on change-proneness

Beibei Wang, Lin Chen*, Wanwangying Ma, Zhifei Chen, Baowen Xu
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

***Abstract*—The dynamic features of programming languages are useful constructs that bring developers convenience and flexibility, but they are also perceived to lead to difficulties in software maintenance. Figuring out whether the use of dynamic features affects maintenance is significant for both researchers and practitioners, yet little work has been done to investigate it. In this paper, we conduct an empirical study to explore whether program source code files using dynamic features are more change-prone and whether particular categories of dynamic features are more correlated to change-proneness than others. To this end, we statically analyze historical data from 4 to 7 years of the development of seven open-source systems. We employ Fisher and Mann-Whitney hypothetical test methods, along with logistic regression model to solve three research questions. The results show that: (1) files with dynamic features are more change-prone, (2) files with a higher number of dynamic features are more change-prone, and (3) *Introspection* is shown to be more correlated to change-proneness than the other three categories in most systems. This innovative work can give some inspirations and references to researchers who are always focusing their eyes on how and why the dynamic features are used. For practitioners, we suggest them to be wary of files with dynamic features because they are more likely to be the subject of their maintenance effort.**

Keywords- dynamic features; change-proneness; Python; empirical software engineering; open-source

I. INTRODUCTION

In recent years, many researchers have shown great interest in the use of dynamic features or dynamic behaviors of programming languages, such as Python, JavaScript and Ruby. Previous works were conducted mainly to discuss whether practitioners are willing to use dynamic features, the main reasons that drive people to use them and how these features are used [1], [2], [3], [4] . Besides, there is a long and ongoing debate about the possible pros and cons of dynamic features in programming languages. Some authors state that dynamic features are of benefit for their flexibility, expressivity and succinctness [5]. For example, the commonly available reflective mechanisms include support for checking available fields/methods, adding and removing fields/methods without the need to restart or rebuild the running program. Others hold the opposite view that the use of these features may hinder software evolution and lead to difficulties in software maintenance. For instance, the use of *eval* endows programmers with the ability to extend applications, at any time, and in almost any way they choose, but it will affect the optimizations that can be applied to programs and significantly limit the kinds of errors that can be caught statically and the security guarantees that can be enforced [4].Hence, it is of great significance to investigate the relation between the use of dynamic features and system maintenance. However, to the best of our knowledge, little work was focused on the effect of dynamic features on program maintenance or evolution, let alone the use of Python dynamic features. Therefore, we make an empirical study on the relation between dynamic features and change-proneness which is well-known to be an indicator of maintenance in the previous study.

Goal. We aim to investigate the effects of 18 Python built-in dynamic features, classified into four broad categories, on the three types of code evolution phenomena. First, we study whether files with dynamic features have an increased likelihood of changing compared to other files. Second, we study whether files with more dynamic features than others are more change-prone. Third, we study the relation between the particular categories of dynamic features and change-proneness.

Contribution. This paper makes three contributions.

- This work is the first one to consider the effect of dynamic features on change-proneness, especially concerning Python language, and thus it will give some inspirations and references for the successors.
- We analyze multiple historical releases of 7 open-source systems to collect the occurrence of 18 Python built-in dynamic features of each file and change information between two versions. The data we gather and publish[1] are useful for the follow-up studies related.
- We get an instructive conclusion from the results of the experiment that although developers are benefit from the flexibility and convenience brought by dynamic features, they should be prudent with them since these features might contribute to more maintenance effort.

The remainder of this article is structured as follows. Section II introduces an overview of related work. Section III provides a description of the 18 Python dynamic features as well as the classification and our detection approach for them. Section IV describes the exploratory study definition and design. Section V presents the study results. Section VI gives a detailed explanation and discussion, along with threats to validity. Finally, Section VII concludes the study and outlines the future work.

II. RELATED WORK

Until now, as far as we know, there has been no study of the relation between dynamic features and change-proneness. Several works studied the usage of dynamic features of various languages, such as JavaScript, Smalltalk and Python, by dynamically or statically analyzing the source code. We will summarize these works as well as works that aimed at relating software quality with factors such as metrics, code smells and language characteristics.

*Corresponding author: Lin Chen; E-mail: lchen@nju.edu.cn
[1]https://github.com/MG1333051/Detailed-Research-Results-.git

(DOI reference number: 10.18293/SEKE2015-097)

Previous research on the dynamic features concerned how to collect them and why and how these features were used in practice. Callaú et al. [1] studied the reflection feature in Smalltalk and found that if a large portion of the usages of dynamic features cannot be refactored, others work around limitations of the programming languages. Richards et al. [4] performed a large-scale study on the use of *eval*, the result of which showed that *eval* was often misused and many uses were unnecessary and could be replaced with equivalent and safer code. Holkner and Harland [7] have conducted a study of the use of 14 dynamic features in the Python programming language. Their study focused on a smaller set of programs and concluded that dynamic features occur mostly in the initialization phase of programs and less so during the main computation. Further, Åkerblom et al. [5] did a similar research to Holkner`s study. They showed that dynamic behaviour is neither buried in library code, nor predominantly occurs at program startup time, which is in slight contrast to the results of Holkner`s study. In our study, we were partly inspired by their classification of dynamic features.

Some studies used metrics as quality indicators, such as Basili et al.'s seminal work [9]. Cartwright and Shepperd [10] performed an empirical study on an industrial C++ system, supporting the hypothesis that classes in inheritance relations are more fault prone. It followed that DIT and NOC metrics [11] could be used to find classes that are likely to have higher fault rates. Some studies chose code smells as predictor of change-proneness. For example, Khomh et al. [12] [13] studied the impact of code smells on software change-proneness and showed that, in their corpus, classes with code smells are more change-prone than others.

Still others concentrated on the effect of programming languages on software quality [14], [15], [16], [17]. For instance, Baishakhi Ray et al. engaged a large scale study of programming languages and code quality in Github. They found that language features, such as static v.s. dynamic typing, strong v.s. weak typing, do have a significant, but modest effect on software quality. Bhattacharya and Neamtiu proposed a novel methodology which controls for development process and developer competence, and evaluates how the choice of programming language affects software quality and developer productivity. Fateman discussed the advantages of Lisp over C and how C itself contributes to the "pervasiveness and subtlety of programming flaws." The author categorized flaws into various kinds (logical, interface and maintainability) and discussed how the very design of C, e.g., the presence of pointers and weak typing, makes C programs more prone to flaws.

Our study does not claim to compare which one is the best predictor of software quality. On the other hand, we are motivated by the previous work concerning the relation between language features and software quality, and are enthusiastic about how dynamic features may influence change-proneness since they are claimed to have an effect on maintenance.

III. PYTHON DYNAMIC FEATURES

In this section, we first briefly introduce the 18 built-in Python dynamic features we focus on. Then we describe the method to collect them.

A. Dynamic Features Selection and Classification

Although there are multiple kinds of dynamic features in Python language, we choose the 18 famous and most often used and investigated [5], [6], [7] features, as shown in TABLE I, which are thought to be representative and are classified as *Introspection*, *Object Changes*, *Code Generation* and *Library Loading*. For brevity, we refer to the Python Reference Manual [8] and present the definition of each classification stated as follows, instead of a description of the individual constructs.

Introspection is a mechanism to treat modules and functions in memory as objects, getting information about them, and manipulating them.

Object Changes is a category of features that can update or change the state of an object, and that can update, add or remove fields in a way that may depend on the program state.

Code Generation is a category of features that can execute code generated or imported in text format during runtime.

Library Loading is a category of constructs that can load or reload arbitrary libraries at runtime, which allows deferring decisions such as what library should be loaded according to user input or underlying hardware.

TABLE I. PYTHON DYNAMIC FEATURES OF FOUR CATEGORIES

Categories				
Introspection		***Object Changes***	***Code Generation***	***Library Loading***
hasattr	isinstance	setattr	eval	__import__
getattr	issubclass	delattr	exec	Reload
callable	type	del	execfile	
globals	vars			
locals	super			

B. Dynamic Features Collection

Previous works presented two popular methods to collect the use of dynamic features. One is to statically analyze the source code to identify the occurrence of a certain kind of dynamic feature, e.g. Callaú O et al. [1] developed a framework in Pharo[2] to trace statically the use of dynamic features of Smalltalk. The other is carried out using trace-based dynamic data collection by instrumenting an interpreter to record runtime data [5]**.** Tracing is able to more precisely describe actual uses of a certain feature than purely static analysis but is sensitive to different paths taken in a program due to input.

In our study, we employ the static collection method instead of the dynamic data collection, because it is difficult to choose representative inputs or interaction strategies that will give acceptable code coverage to figure out all files with or without dynamic features. The specific code analysis and data collection process are supported by a static analysis tool *Understand*[3].For each version of a system, we first filter non-Python source files by using the *Python Strict* option in *Understand* to dispose of files unrelated and then build an intermediate database which stores information of entities (function, variable, file, class, attribute et al.), the call graphs among these entities and so forth. After that, we write Perl scripts to invoke *Understand* APIs to mine all program points that use the built-in dynamic features from the database. The algorithm contains three steps:

[2]http://www.pharo-project.org
[3]https://scitools.com/

TABLE II. SUMMARY OF THE CHARACTERISTICS OF THE ANALYZED SYSTEMS

Project	Releases (number)	Duration	Files	LOCs	Description
Boto	2.0-2.28.0 (6)	2011.07-2014.04	217-617	29,246-104,967	interfaces to Amazon Web Services
Bzr	1.2-2.5.0 (9)	2008.02-2012.03	585-830	148,183-263,454	version control system
Django	1.0-1.6 (7)	2008.09-2013.11	956-1872	83,136-165,184	high-level Python Web framework
Matplotlib	0.99.0-1.3.1 (6)	2009.08-2013.10	767-1677	99,934-163,780	library for 2D plotting
Numpy	1.0.4-1.6.2 (8)	2007.12-2012.08	255-398	58,866-119,479	library for mathematics, science, engineering
Scipy	0.7.0-1.13.2 (8)	2009.02-2013.12	419-510	91,479-149,471	library for mathematics, science, engineering
Tornado	1.0.0-3.2.1 (8)	2010.07-2014.05	42-97	10,915-22,095	high-level Python Web framework

1) Firstly, for each function called in a database, the algorithm checks whether it reflects one of the analyzed dynamic features except for *del*, simply by comparing their names. If it matches one, find out the name of the file that uses this function, and thus the number of the matched dynamic features in this file is increased by one.

2) Secondly, for each lexeme in a file recognized by *Understand*, the algorithm checks whether its token is a keyword and its text is equal to *del*. If it is, then record the file name and increase the number of the *del* in this file.

3) Thirdly, for a kind of dynamic feature that does not appear in a file, the algorithm sets the number of that dynamic feature in the file to zero.

4) Finally, the algorithm makes a two-dimensional table stored in .csv format for the subsequent data analysis, which saves all of the file names of a system and the number of each dynamic feature used in every file.

IV. STUDY DEFINITION AND DESIGN

Section four starts with an explanation of how to get change information of each file. Then it presents an introduction of the target systems. After that, it elaborates the research questions and the analysis methods for solving each research question.

A. File Change Information

In the experiment analysis, we need the change information of each file, specifically whether the file is changed or not. To acquire such data, we first write a Perl script to invoke the Linux system command '*diff*' which can be used to compare two arbitrary text files. The execution of the script can generate a formatted difference report textfile that records the position of all the changes and the number of changed lines (added, modified or deleted). Then by writing another script to mine the formatted difference report, we can easily get change data of each file and store them in .csv format likewise. Furthermore, for files that appear in the former version but disappear in the latter version, we identify them as changed files.

B. Data Sets

The context of this study consists of the change history and dynamic features of 7 most famous open-source projects, which have a different size and belong to different domain. For each target system, we regularly choose releases in the interval of 4 to 12 months. Characteristics of the analyzed projects are shown in TABLE II, and the more detailed data are published online[1]. On every considered release, we gather the change information and dynamic features of each file, depending on the methods mentioned earlier.

C. Research Questions

Based on the data collected from the above systems, our study aims to answer 3 research questions.

- RQ1: What is the relation between dynamic features and change-proneness? More specifically, we explore if files with dynamic features are more change-prone than others by testing the null hypothesis: H_{01}: the percentage of files exhibiting at least one change between two releases does not significantly differ between files with dynamic features and other files.
- RQ2: What is the relation between the number of dynamic features in a file and its change-proneness? We analyze whether files with a higher number of dynamic features are more change-prone than others by testing the null hypothesis: H_{02}: the number of dynamic features in change-prone files is not significantly higher than the number of dynamic features in files that do not change.
- RQ3: What is the relation between particular categories of dynamic features and change-proneness? Since, we are also interested to evaluate whether particular categories of dynamic feature contribute more than others to changes by testing the null hypothesis: H_{03}: files with particular categories of dynamic features are not significantly more change-prone than other files.

D. Analysis Methods

To answer RQ1, we test whether the proportion of files undergoing (or not) at least one change significantly varies between files with dynamic features and other files by using Fisher's exact test [18].This test is appropriate for categorical data that result from classifying objects in two different ways and is used to examine the significance of the association (contingency) between the two kinds of classification. To apply the test, we divide the files of each release into four groups, that is, (1) files undergoing at least one change and with at least one dynamic feature; (2) files undergoing at least one change but with no dynamic feature; (3) files undergoing no change but with at least one dynamic feature; (4) files neither changing nor using dynamic feature. In addition, we compute the odds ratio (*OR*) [18]. The *OR* is the ratio of the odds *p* of an event occurring in one group, i.e., the odds that files with dynamic features underwent a change (experimental group), to the odds *q* of it occurring in another group, i.e., the odds that files with no

dynamic features underwent a change (control group), more intuitively: $OR=\frac{p/_{1-p}}{q/_{1-q}}$. An *OR* greater than 1 indicates that changes are more likely to happen in files with dynamic features, while an *OR* less than 1 means that changes are more likely to happen in files without dynamic features. If odds ratio equals to 1, the event is equally likely in both samples.

In RQ2, we use the Mann-Whitney test to compare the number of dynamic features in change-prone files with the number of dynamic features in non-change-prone files. The Mann-Whitney test is a non-parametric test that does not require any assumption on the underlying data distributions, and thus is suitable for our experiment. Other than testing the hypothesis, it is of practical interest to estimate the magnitude of the difference of the number of dynamic features in files with and without changes, thus we use the Cohen`s *d* effect size [18]. A *d* greater than 0 indicates that the number of dynamic features are more in changed files than in not changed files, and less than 0, the contrary. It is worth mentioning that the effect size is often considered small for $0.2 \leq |d| < 0.5$, medium for $0.5 \leq |d| < 0.8$ and large for $|d| \geq 0.8$. For RQ2, we consider the files change or not as the independent variable, and the number of dynamic features in files as the dependent variable.

In RQ3, to relate change-proneness with the presence of particular categories of dynamic features, we use a logistic regression model which is widely used in many studies, e.g., [12], [19], to deal with similar problems. In the logistic regression model, the dependent variable is commonly a dichotomous variable and, thus, only two values {0, 1}, i.e., in this article changed or not. The multivariate logistic regression model is based on the formula:

$$\pi(X1, X2, \ldots\ldots, Xn) = \frac{e^{\beta 0+\beta 1 \cdot X1+\ldots+\beta n \cdot Xn}}{1+e^{\beta 0+\beta 1 \cdot X1+\ldots+\beta n \cdot Xn}}$$

where (a) *Xt* are characteristics describing the modelled phenomenon, in our case, the number of dynamic features of category t a file contains; (b) βt are the model coefficients; and c) $0 \leq \pi \leq 1$; the closer the value is to 1, the higher is the likelihood that the file undergoes a change. For each category of dynamic features, we count the number of times that, across the analyzed releases of a target system, the *p*-values obtained by the logistic regression are significant. If files participating in a specific category of dynamic features are more likely to change in more than 75% of the releases of a target system, then we say that this category of dynamic features has a significant impact on increasing the change-proneness in this system.

V. Study Results

In this section, we present the results of our empirical study which are further discussed in section six. More detailed results and raw data are available online[1].

A. RQ1: Dynamic Features and Change-Proneness

TABLE III reports the results of Fisher's exact test and *OR* values when testing H_{01}. For each target system, it presents the number of all the releases that are analyzed and the number of releases whose *p*-values of Fisher's test are significant (p-values<0.05).To be specific, six of seven projects turn out to be significant for more than 75% of their releases, and three projects even prove to be significant for all the releases analyzed. The only outlier is Tornado, five of eight releases turn out to be significant. In summary, although the results sometimes depend on systems analyzed, we can reject H_{01}, i.e., the percentage of files exhibiting at least one change between two releases does significantly differ between files with dynamic features and other files. Regarding the *ORs* of significant releases, they vary across systems and, within each system, across releases. In 75% of the releases of six systems, the *ORs* for files with dynamic features to change are two times higher or more than for files without dynamic features and thus odds to change is in general higher for files with dynamic features. In very few releases of some systems, as highlighted, *ORs* are close to 1,i.e, the odds are even that a file with a dynamic features changes or not.

We therefore conclude that, in most cases, there is a negative relation between dynamic features and change-proneness: a greater proportion of files participating in dynamic features change comparing to other files. Developers should be wary of files with dynamic features, because they are more likely to be the subject of their maintenance effort.

B. RQ2: Number of Dynamic Features and Change-Proneness

TABLE IV presents results of the Mann-Whitney two-tailed test and Cohen`s *d* effect size of the target systems, with the purpose of comparing the number of dynamic features in files that changed or not. More than 75% of the releases of all projects, show significant *p*-values with relatively small to medium effect sizes, except for Tornado, where only 4 out of 8 releases are significant but with a medium effect size. Moreover, the releases that prove not to have significant *p*-values confirm the findings from RQ1 regarding the limited relation of dynamic features with change-proneness for these releases. It is worth mentioning that *p*-value of boto-2.6.0 is significant (*p*-value=0.02) in RQ2

TABLE III. Summary of Fisher test results and OR values for each target system

Project	Number of analyzed releases	Number of significant p-values	Percent of significant p-values	OR					
				Max	*Min*	*Mean*	*25% quartile*	*50% quartile*	*75% quartile*
Boto	6	5	83.3%	4.18	1.97	2.83	2.04	2.15	3.96
Bzr	9	8	88.9%	4.77	2.05	3.00	2.44	2.82	3.33
Django	7	7	100%	10.13	**1.38**	5.88	3.47	4.48	9.53
Matplotlib	6	6	100%	27.07	1.61	8.71	3.78	5.30	13.13
Numpy	8	8	100%	4.77	2.19	3.47	2.71	3.54	4.31
Scipy	8	6	75%	4.04	1.50	2.87	1.69	3.30	3.91
Tornado	8	5	62.5%	8.70	3.94	5.96	4.08	6.41	7.61
Sum	52	45	86.5%	-	-	-	-	-	-

TABLE IV. SUMMARY OF MANN-WHITNEY RESULTS AND COHEN`S D FOR EACH TARGET SYSTEM

Project	Number of analyzed releases	Number of significant p-values	Percent of significant p-values	Cohen`s d					
				Max	*Min*	*Mean*	*25% quartile*	*50% quartile*	*75% quartile*
Boto	6	6	100%	0.60	0.16	0.39	0.27	0.38	0.56
Bzr	9	8	88.9%	0.45	-0.01	0.34	0.30	0.38	0.44
Django	7	7	100%	0.61	0.07	0.35	0.28	0.36	0.41
Matplotlib	6	6	100%	0.82	0.12	0.45	0.21	0.40	0.73
Numpy	8	8	100%	0.55	0.05	0.38	0.29	0.43	0.51
Scipy	8	6	75%	0.55	0.33	0.44	0.38	0.45	0.50
Tornado	8	4	50%	0.75	0.54	0.64	0.56	0.64	0.73
Sum	52	45	86.5%	-	-	-	-	-	-

but not significant (*p*-value=0.14) in RQ1, yet we consider it a tolerable abnormal phenomena that does not affect the whole results. In summary, the results of most releases support that change-prone files are those with a higher number of dynamic features and thus we can reject $H_{02.}$

C. RQ3: Categories of Dynamic Features and Change-Proneness

TABLE V summarizes the results of the logistic regression for the correlations between change-proneness and the different categories of dynamic features. In particular, the table presents the number of analysed releases for which each categories of dynamic features is significant in the logistic regression model. Boldface indicates significant p-values for at least 75% of the releases in each system. Following our analysis method of RQ3 in section four, it is noticed that *Introspection* is shown to be significantly correlated to change-proneness in 5 target systems, and that *Library Loading* only has impact on Numpy project. However, for Boto and Tornado, there are not enough releases to support the relation between any category of dynamic features and change-proneness. Therefore, we can partly reject H_{03} for *Introspection* and *Library Loading* depending on the results observed. On the whole, although only 5 of 7 analyzed systems reject H_{03}, we can conclude that there are categories of dynamic features which are more related to others to change-proneness in most cases and that the relation between particular categories of dynamic features and change-proneness cannot be completely ignored. What is more, the *Introspection* category deserves extra attention for it turns out to be more related to change-proneness than others.

TABLE V. NUMBER OF RELEASES WHERE EACH CATEGORY OF DYNAMIC FEATURES SIGNIFICANTLY CORRELATES WITH CHANGE-PRONENESS.

Project	Number of analyzed releases	Proneness to Change of each category of Dynamic features			
		Introspection	*Object Changes*	*Code Generation*	*Library Loading*
Boto	6	3	2	-	-
Bzr	9	**7**	4	1	-
Django	7	**6**	2	1	1
Matplotlib	6	**5**	3	2	1
Numpy	8	**6**	-	2	**6**
Scipy	8	**6**	-	2	-
Tornado	8	3	1	-	-

VI. DISCUSSION

We now discuss the implications of the results reported in section five, along with threats to validity.

A. Discussions and Implications

In this study, we investigate the impact of 18 built-in Python dynamic features on file change-proneness. As analyzed in section five, the results show that files with dynamic features (and, in particular, those with a higher number of dynamic features) are significantly more change-prone than others in most releases of the analyzed systems, except for Tornado. And dynamic features of *Introspection* are more related to file change-proneness than the other three categories. Based on these results, we can get some useful implications for both research and practice.

For the research community, this work is the first one to focus on the relation between dynamic features and maintenance. The negative relation between dynamic features and change-proneness promotes further investigations to be conducted on the relation between dynamic features and other maintenance related factors, such as fault-proneness. In sum, our study inspires researchers to turn their attention from how and why to use dynamic features to the effect that these features have on maintenance. Additionally, we suggest that more work should be focused on the category of dynamic features that affect change-proneness most, in this work, the *Introspection* category, and on how and why this kind of feature can be constructed, in order to improve the quality of software and help us better understand dynamic features as well.

For practice, we suggest that developers should be cautious when using dynamic features, especially the *Introspection*, because the presence of these features may lead to the maintenance effort and cost. As for quality assurance personnel, they need to pay extra attention to files with more dynamic features, since these files may contribute to more maintenance problems.

In addition to the foregoing, it is noticed that Tornado does not exhibit an overwhelming significant relation (percent of significant *p*-values $\geqslant$75%) of all the releases even if in one of the three RQs. We deduce the reason for this fact lies in the minor number of files of each release ranging from 42 to 97, while file number of the other systems varies from hundreds to thousands.

B. Threats to Validity

Internal threats in this work mainly concern whether the hypothesis testing methods are properly used. Although in practice the Fisher's exact test is often employed when sample sizes are small, it is also valid for all the sample sizes. Also, we

choose the non-parametric tests that do not require making assumption about the data set distribution. To build the logistic regression model, it is important to discard the independent variables that are highly correlated to each other. We eliminate such a threat by calculating the Spearman rank correlation coefficient between any two different categories of dynamic features. As expected, the results[3] show that no two categories of dynamic features are highly correlated (Spearman rank correlation coefficient is higher than 0.8), and thus it is no need to exclude any of the independent variables in our experiment.

Threats to external validity concern the possibility to generalize our findings. Although we have tried our best to limit such a threat and make the results general by choosing 7 open-source systems of 5 different problem domains, as shown in TABLE I, and by covering most of the built-in Python dynamic features that are representative in each of the categories, yet the generalization still requires further case studies including a large number of Python systems from various domains and more dynamic features as well. Besides, since covering all historical versions for one project is a hard work, we select them regularly by an interval of 4 to 12 months, which is a reasonable way.

Construct validity threats concern the relation between theory and observation. In our context, they are mainly due to errors introduced in measurements. In this work, the count of changes occurred to files is based on comparing the difference of files with the same name but from two versions. We are just interested to check whether a file changes or not, rather than quantifying the amount of change, which is however possible based on rules in [20] and could be investigated in the future work. In our detection algorithm, we ignore dynamic features appearing in annotated codes. But we consider it does not influence our results, for these circumstances are rare and are often used for illustration purpose not for realizing functions.

VII. Conclusions and Future Work

In this paper, we explore how the use of dynamic features affects file change-proneness. The whole study is undertaken by choosing 18 most often used and studied Python dynamic features [5], [6], [7] and 7 famous open-source Python systems from Github and SourceForge online repositories. We find that files with dynamic features are significantly more likely to be the subject of changes, than other files. We also show that dynamic features of *Introspection* are more likely to be of concern during evolution. This exploratory study supports, within the limits of the threats to its validity, the conjecture in the literature that dynamic features may have a negative impact on software evolution. Depending on the results observed, we suggest practitioners that they should be cautions of treating systems with a high prevalence of dynamic features during development and maintenance, because those systems are likely to be more change-prone: therefore, the cost-of-ownership of such systems will be higher than for other systems. Additionally, we call on researchers to pay more attention to dynamic features of other languages concerning their impacts on software quality and on the root causes of their negative impact, on the basis of our work.

In the future work, we will replicate this study on more systems and with more dynamic features considered to validate the above-mentioned findings. Further, we are interested to relate dynamic features to other phenomena such as the fault-proneness.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (61472175, 61170071, 61472178), and the National Natural Science Foundation of Jiangsu Province (BK20130014). I express my sincere gratitude to all the teachers and students who make contributions to this work.

References

[1] Calla ú O, Robbes R, Tanter É, Röthlisberger D. How (and why) developers use the dynamic features of programming languages: the case of Smalltalk. *Empirical Software Engineering* 18.6 (2013): 1156-1194.

[2] An, J. H. D., Chaudhuri, A., Foster, J. S., & Hicks, M. (2011). Dynamic inference of static types for ruby (Vol. 46, No. 1, pp. 459-472). *ACM*.

[3] Richards, G., Lebresne, S., Burg, B., & Vitek, J. (2010, June). An analysis of the dynamic behavior of JavaScript programs. In *ACM Sigplan Notices* (Vol. 45, No. 6, pp. 1-12).

[4] Richards, G., Hammer, C., Burg, B., & Vitek, J. (2011). The eval that men do. In *ECOOP 2011–Object-Oriented Programming* (pp. 52-78).

[5] Åkerblom, B., Stendahl, J., Tumlin, M., & Wrigstad, T. (2014, May). Tracing dynamic features in python programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (pp. 292-295).

[6] Tratt, L. (2009). Dynamically typed languages. *Advances in Computers*, *77*, 149-184.

[7] Holkner, A., & Harland, J. (2009, January). Evaluating the dynamic behaviour of Python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91* (pp. 19-28).

[8] G. van Rossum and F.L.Drake, "PYTHON 2.6 Reference Manual", CreateSpace, Paramount, CA, 2009.

[9] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *TSE*, 22(10):751–761, 1996.

[10] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *TSE*, 26(8):786–796, August 2000.

[11] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, *20*(6), 476-493.

[12] Khomh, F., Di Penta, M., & Gueheneuc, Y. (2009, October). An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on* (pp. 75-84). IEEE.

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

[14] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar T Devanbu. A Large Scale Study of Programming Languages and Code Quality in Github. *FSE'14*, 16–22, 2014.

[15] S. Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. *OOPSLA '10*, 22–35, 2010.

[16] Fateman, R. (2002). Software fault prevention by language choice: Why C is not my favorite language. *Advances in Computers*, *56*, 167-188.

[17] Pamela Bhattacharya and Iulian Neamtiu. Assessing Programming Language Impact on Development and Maintenance: A Study on C and C++. *ICSE'11*, 21–28, 2011.

[18] D. Sheskin. Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition). Chapman & All, 2007.

[19] Hosmer Jr, D. W., & Lemeshow, S. (2004). *Applied logistic regression*. John Wiley & Sons.

[20] Yuming Zhou, Hareton Leung and Baowen Xu. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and Change-Proneness. *TSE*, 607-623, 2009.