METRIC-BASED CODE SMELL DETECTION TECHNIQUE FOR PYTHON SOFTWARE

SAMAILA JA'AFARU LEEMAN

FSKTM 2019 55

METRIC-BASED CODE SMELL DETECTION TECHNIQUE FOR PYTHON SOFTWARE

By

SAMAILA JA'AFARU LEEMAN

Thesis Submitted to the School of Graduate Studies, Universiti Putra Malaysia, in Fulfilment of the Requirements for the Degree of Master of Science

March 2018

Abstract of thesis presented to the Senate of Universiti Putra Malaysia in fulfilment of the requirement for the degree of Master of Science

METRIC-BASED CODE SMELL DETECTION TECHNIQUE FOR PYTHON SOFTWARE

By

SAMAILA JA'AFARU LEEMAN

March 2018

Chairman : Norhayati Mohd Ali, PhD
Faculty : Computer Science and Information Technology

In software life cycle, maintenance is reported to cost between 80% and 90% of the total software cost. Instead of improving, software quality can deteriorate if maintenance of software is not carried out in line with design principles. A term, "code smell" has been framed to refer to software constructs that deviates from design principles' implementation. Refactoring, a process of removing code smells has been recommended to improve maintainability qualities. However, for refactoring to be applied, code smells must be detected in the location where they exist. While research on code smells in Java and C++ programmed codes is mature, similar research in Python software is scarce. There are few literatures that show metric-based approaches have been used to detect code smells in Python Software. However, the techniques require selection of metrics and a tedious calibration of threshold upon which detection is done. Existing metric-based techniques anticipates parser to handle only one version of Python though Python 2 and 3 are popular and end of life for Python 2 has been announced. This research proposes an enhanced metric-based technique named PySTect that extracts structural information from parsed models of any version of Python software and translates them into metrics for detection of code smells. This research is divided into four phases. The first phase involves reviewing the literatures on code smells detection techniques, parser models, and code smells in Python software. In the second phase, details of conceptual and architectural design of the metric-based technique are presented. Thirdly, PySTect technique is implemented and evaluated in three experiments. In the first experiment, precision and recall of 100% and 96% were recorded which indicates the effectiveness of the PySTect technique in detecting five code smells. In the second and third experiments 1,167,180 lines of code from 13 open source Python projects are analysed. Results of the second experiment shows codes smell increases with evolution of Python software projects and the third experiment indicates the most dominant smells in Python SDKs are Lazy Class, Improper Method Declaration, a Python specific smell and large class, in that order. This research work has proposed an enhanced metric-based detection technique for Python software that analyses Python codes irrespective of its versions and extracts thresholds from good designed codes saving developers effort to calibrate threshold. It has contributed in specifying and detecting a new code smell, Improper Method Declaration and also confirmed that dynamic programming language suffers from large class, lazy class, long method and long parameter list like static languages. Lastly, this research found that code smells increases in Python programs with evolution and, lazy class and improper method declaration should be prioritized in SDK domain.

Abstrak tesis yang dikemukakan kepada Senat of Universiti Putra Malaysia sebagai memenuhi keperluan untuk ijazah Master Sains

TEKNIK PENGESANAN *Code Smell* BERASASKAN METRIK UNTUK PERISIAN *Python*

Oleh

SAMAILA JA'AFARU LEEMAN

Mac 2018

Pengerusi : Norhayati Mohd Ali, PhD
Fakulti : Sains Komputer dan Teknologi Maklumat

Di dalam kitaran hayat perisian, anggaran bagi kos penyelengaraan adalah di antara 80% hingga 90% daripada jumlah kos perisian. Jika penyelengaraan perisian tidak dilaksanakan mengikut prinsip reka bentuk, kualiti perisian tidak akan bertambah baik sebaliknya ia akan terus merosot. Istilah "*code smell*" telah diperkenalkan bagi merujuk kepada pembangunan perisian yang tidak mengikut pelaksanaan prinsip reka bentuk. Pemfaktoran semula, adalah satu proses pembuangan *code smell* yang dicadang bagi meningkatkan kualiti penyelenggaraan. Walau bagaimanapun, pemfaktoran semula hanya boleh dilaksanakan sekiranya lokasi *code smell* tersebut dijumpai. Walaupun kajian *code smell* di dalam kod aturcara Java dan C++ sudah matang, tetapi ia masih kurang bagi perisian Python. Terdapat beberapa literatur yang menunjukkan pendekatan berasaskan metrik telah digunakan untuk mengesan *code smell* dalam Perisian Python. Walau bagaimanapun, teknik ini memerlukan pemilihan bagi metrik dan penentukuran ambang yang membosankan di mana pengesanan dilakukan. Teknik berasaskan metrik yang sedia ada menjangkakan *parser* untuk mengendalikan hanya satu versi Python walaupun Python 2 dan 3 adalah popular dan akhir hayat untuk Python 2 telah diumumkan. Kajian ini mencadangkan satu teknik berasaskan metrik yang dipertingkatkan yang dinamakan PySTect untuk ekstrak maklumat struktur dari model penghurai bagi mana-mana versi perisian Python dan diterjemahkan ke dalam metrik untuk pengesanan *code smell*. Kajian ini terbahagi kepada empat fasa. Fasa pertama melibatkan kajian literatur ke atas teknik pengesanan *code smell*, model *parser* dan *code smell*dalam perisian Python. Di dalam fasa kedua, reka bentuk konseptual dan seni bina bagi teknik berasaskan metrik diperincikan. Untuk fasa ketiga, teknik ini akan dilaksana dan diuji menggunakan bahasa aturcara Phython 3 di dalam satu alat prototaip yang dipanggil PySTect. Pada fasa terakhir, tiga eksperimen dijalankan untuk menjawab tiga soalan kajian yang dicadangkan di dalam tesis ini. Di dalam eksperimen pertama, 100% ketepatan dan 96% perolehan telah direkodkan dan ia menunjukkan keberkesanan teknik PySTect dalam mengesan lima *code smell*. Untuk eksperimen kedua dan ketiga, sebanyak 1,167,180 baris kod daripada 13 projek Python dari sumber terbuka dianalisis. Keputusan eksperimen kedua menunjukkan bilangan *code smell* telah meningkat seiring dengan evolusi perisian Python. Eksperimen ketiga pula menunjukkan *code smell* yang paling utama dalam SDK Python adalah *Lazy Class*, *Improper Method Declaration*, *code smell* khas Python dan *Large Class*. Kajian ini mencadangkan teknik pengesanan berasaskan metrik untuk perisian Python di mana ia akan menganalisa kod Python tanpa mengira versi dan ia akan mengekstrak ambang dari kod yang mempunyai reka bentuk yang baik. Kesannya, ia menjimatkan masa pembangun perisian dalam menentukan nilai ambang. Kajian ini telah menyumbang kepada cara menentukan dan mengesan *code smell* yang baru, kesilapan pengisytiharan kaedah dan mengesahkan bahawa bahasa pengaturcaraan dinamik mempunyai masalah berkaitan kelas besar, kelas malas, kaedah yang panjang dan senarai parameter yang panjang seperti bahasa statik. Akhir sekali, penyelidikan ini mendapati bahawa bilangan *code smell* terus meningkat dalam program Python seiring dengan evolusi program. Selain daripada itu, *Lazy Class* dan *Improper Method Declaration* harus diutamakan dalam domain SDK.

ACKNOWLEDGEMENTS

Research of this nature cannot be accomplished without the guidance and support of experienced hands. In recognition of this, may I acknowledge the contribution of my supervisory committee who held my hands along this journey. It was a great privilege for me to work under the expert guidance of Assoc. Professor Rodziah Binti Atan and Dr. Khaironi Yatim Sharif. I wish to single out the Chairman of the research committee, Dr. Norhayati Mohd Ali for horning my research skills. Her kind words, constructive criticism and patient allowances for my weaknesses were great sources of inspiration and encouragement for me.

I want to thank members of my research group especially members of the Software Engineering Lab. Here, Osman, Luqman and Ididi Williams easily comes to mind. May I also acknowledge the support of Dr. Aliyu and Engr. Maryam Ahmed, Mr. Kabu, Yapilami, Pudza, Timothy, Irene, Eevon and others too numerous to mention. Their friendship and inputs made great impact on my rewarding experience at UPM.

Let me now thank my dearest wife, Myhrt who filled every gap during my long absence. She provided succour and encouragement to the children, Shekwolaye, Shay’amu, Salman to bear my absence. I will forever be indebted for their patience, endurance and many sacrifices. May I also express appreciation to my Mama, Ladi and my siblings who were left to mourn the transition of my dad in my absence. I must thank Engr. Hassan Odediran, Hon. Marcus Yari, my friends and pastors in Nigeria and Malaysia who provided me support both spiritual and temporal during the period of my studies.

Special thanks to management of Kaduna Polytechnic for granting me the permission and support to travel abroad for the studies.

Now to the King Eternal, Immortal, Invisible, the only God, be honour and glory for ever and ever. Amen.

This thesis was submitted to the Senate of Universiti Putra Malaysia and has been accepted as fulfilment of the requirement for the degree of Master of Science. The members of the Supervisory Committee were as follows:

Norhayati Mohd Ali, PhD
Senior Lecturer
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia
(Chairman)

Rodziah Atan, PhD
Associate Professor
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia
(Member)

Khaironi Yatim Sharif, PhD
Senior Lecturer
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia
(Member)

____
ROBIAH BINTI YUNUS, PhD
Professor and Dean
School of Graduate Studies
Universiti Putra Malaysia

Date:

Declaration by Members of Supervisory Committee

This is to confirm that:

- the research conducted and the writing of this thesis was under our supervision;
- supervision responsibilities as stated in the Universiti Putra Malaysia (Graduate Studies) Rules 2003 (Revision 2012-2013) were adhered to.

Signature:	
Name of Chairman of Supervisory Committee:	Dr. Norhayati Mohd Ali

Signature:	
Name of Member of Supervisory Committee:	Associate Professor Dr. Rodziah Atan

Signature:	
Name of Member of Supervisory Committee:	Dr. Khaironi Yatim Sharif

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ABBREVIATIONS

BDFL	Benevolent Dictator For Life
CM	Class Method
F_arg	First Argument
IMD	Improper Method Declaration
IDE	Integrated Development Environment
JSNose	JavaScript Nose
LOC	Lines of Code
MDec	Method Decorated
MDecl	Method Declaration
NOA	Number of Attribute
NOM	Number of Methods
NOP	Number of Parameter
PEP	Python Enhancement Proposal
PYPI	Python Packages Index
Pysmell	Python Smell
Pystect	Python Smell Detector
SDK	Software Development Kit
SM	Static Class
Us_f_arg	Used First Argument

CHAPTER 1

INTRODUCTION

1.1 Background

In software development life cycle, cost of software maintenance, is between 80% and 90% of the total software cost (Bavota, De Lucia, Marcus, & Oliveto, 2013; Bennett & Rajlich, 2000). Maintenance in software engineering refers to the modifications of software after it has been deployed. Some of maintenance activities include, addition of functionalities, correction of bugs, adaptation to new technologies and environment, and changes in development rules and practices in the cause of software evolution. These activities over time, tends to negatively affect the software qualities such as changeability, understandability, readability and testability of software (Tripathy & Naik, 2015). The degradation of these qualities accumulates into maintainability issues and consequently raising cost of maintenance. While maintenance activities are unavoidable, carrying out the activities without adhering to design principles or design patterns are at the roots of factors that trigger degradation of software qualities (Ouni, Kessentini, Member, & Kessentini, 2014). If this quality degradation of software is allowed to continue, it may ultimately lead to software project failure as cost of maintenance becomes unaffordable.

To reverse software degradation and consequently mitigate against imminent failure, restructuring of the software is recommended to bring the software in compliance with design heuristics and principles. In Object Oriented Design paradigm, restructuring is referred to as refactoring. Refactoring is defined by Fowler (Fowler, Beck, Brant, Opdyke, & Roberts, 1999) as the rearrangement of software structure without changing its observable characteristics.

However, for refactoring to occur, fragment of codes in need of refactoring must be identified at a location since they are pervasive. Kent Beck is reported in Fowler et al. (1999) as the first to use the term Bad Code Smell to refer to fragment of codes that violates basic Object-Oriented Design principles. Code Smell therefore will require to be detected in a code base for refactoring to be applied (Bassey, Dladlu, & Ele, 2016). This has severally been suggested by researchers that detection of refactoring candidates in object oriented code bases is the first step towards applying refactoring.

Manual inspection of codes has been a historic practice in software engineering for identifying fragments of codes that require attention (Moonen, 2002). Manual inspection, however does not scale with size especially now that code bases are burgeoning into thousands of lines. Manual detection of patterns in codes that deviate from design principles is not only laborious, unrepeatable and prone to errors, it also requires experience to track (Murphy-Hill, Zimmermann, Bird, & Nagappan, 2014). There are a number of automated or semi-automated tools that have shown good promise in assisting developers using Object Oriented Languages with code smell detection (Fernandes, Oliveira, Vale, Paiva, & Figueiredo, 2016). However, majority of these tools do not support developers working with dynamic languages such as Python (Chen, Chen, Ma, & Xu, 2016; Fard & Mesbah, 2013). Research on code smells in dynamically-typed languages is scarce and if attention is not given to it, it might pose great challenge to the future of promising programming language development.

In this research, an enhanced metric-based technique is proposed for the detection of code smells in Python software. The technique compares with previous techniques proposed by (Chen et al., 2016; Fard & Mesbah, 2013; Moha, Guehenuec, Meur, Laurence, & Tiberghien, 2010) in specification and detection of code smell.

1.2 Problem Statement

Research has shown that code smell is capable of negatively affecting software quality such as understandability, readability, reusability testability and modifiability (Abbes, Khomh, Guéhéneuc, & Antoniol, 2011; Sz, Antal, Nagy, Ferenc, & Gyimóthy, 2017). Among many techniques for detection of code smells, literatures have indicated that metric-based technique is most reliable (Yamashita & Moonen, 2013b). Research by Chen et al., (2016) detects code smells in Python programs using Pysmell, a metric-based technique. However, the detection process is not transparent, threshold upon which detection is carried out is manually computed and there is no indication the technique can analyse various versions of Python. The technique uses a syntax tree analyser to collect metrics for detection but it does not compute the criteria by which code smells are detected. This saddles the user with the responsibility to manually determine the detection threshold which leaves so much to the whims or experience of the user as calibration of appropriate threshold requires some expertise.

Code smell detection is critical to refactoring and is evidenced by the attention researchers have accorded code smell in DÉCOR (Moha, Duchein, Guehenuec, & Meur, 2010), JDeodorant (Fokaefs, Tsantalis, Stroulia, & Chatzigeorgiou, 2012), inCode (Yamashita & Moonen, 2013b), TrueRefactor (Griffith, Wahl, & Izurieta, 2011). Previous studies have indicated that Large Class, Long Method, Lazy Class, dead code and Long Parameter List are predominant code smells in statically-typed languages (Chen et al., 2016; Fard & Mesbah, 2013; Moha, Duchein, et al., 2010). However, works by Chen et al., (2016) and Fard & Mesbah (2013) did not provide information on which code smell is predominant in specific software domain of Python software such as SDK. Knowledge about domain-specific code smell can assist Python developers to pay attention to certain pitfalls that deteriorate software quality as a results of code smells.

Python is an object-oriented language whose functions are first class objects. This refers to Pythons feature in which functions have attributes and can be referenced, passed as argument of another function and be assigned to variables (Lott, 2014). Though these features of Python offers so much flexibility, it opens up opportunity for breeding of code smells capable of ruining software qualities especially as software evolve with every maintenance cycle (Yamashita & Moonen, 2013a; Fard & Mesbah, 2013). There is scarcity of information on how code smell density changes as Python software undergo evolution.

1.3 Research Questions

The purpose of this research is to answer the following questions:

1. How effective is the proposed automated metric-based technique in detecting code smells in a parser model of Python software?
2. How does code smell density change with evolution in Python codes?
3. What are the predominant code smells in SDK domain?

1.4 Research Objectives

The objective of this research is to detect code smells in Python source codes using the proposed automated metric-based technique. The specific objectives are itemised as follows:

1. To propose an automated metric-based technique (PySTect) to detect code smells in Python parser model.
2. To evaluate the density of code smell changes with software evolution.
3. To determine predominant code smells in SDK domain via four Python open source projects and PySTect tool.

1.5 Scope of the Research

This research proposes an automated metric-based technique for detection of code smell in Python software. There are many code smells discussed in literature however, this work focused on five code smells among which are four most researched code smells, namely; Large Class, Lazy Class, Long Method and Long Parameter List. The fifth code smell, Improper Method Declaration is a newly defined code smell in this research. The famous four code smells were selected to be detected in this thesis to provide information whether Python software similarly suffers from the five code smells. The automated metric-based technique in this thesis detects code smell statically and is validated on Python software. However, the technique can be extended to analyse codes in other dynamic languages with little modification in the presence of the language's parser model.

1.6 Significance of the Research

The research discussed in this thesis contributes to the field of software engineering particularly in the area of software maintenance. Code smell detection in software is the primary step to refactor source code. The main contributions from this research are as follows:

1. This research provides an automated metric-based technique for detection code smells in Python software. The technique incorporates a version converter to allow for detection of code smells in Python software irrespective of its version. The technique automates extraction of metrics and threshold values for detection of five code smells saving developers' effort in calibration of threshold. The technique can be extended with little modification to detect code smells in other dynamic languages other than Python.
2. This research provides an architecture to specify and detect code smell in Python software. Four major components are defined: Input and cleaning of data, parser component, smell detection, and detection report components.
3. The automated metric-based technique is implemented into a prototype tool for code smell detection in Python software called PySTect as a proof-of-concept of the technique. The prototype is evaluated using three experiments. Results from the evaluation shows that Python software suffers from code smells and code smell density increases with the software evolution.
4. The result also shows that Python SDKs suffer more from Lazy Class, Improper Method Declaration and Large Class smells than from Long Method and Long Parameter List. This information will guide Python SDK developers prioritize this code smells.

1.7 Organization of the Thesis

This work is broadly structured into six chapters as outlined in the following:
In chapter one, the general overview of the thesis is presented. The problems the research sets out to solve are reduced to research questions. The research questions are transformed into objectives of the research. The chapter closes with stating the scope and itemising significance of the research.

In chapter two, the related literatures that form the body of knowledge of the domain is reviewed to gain deep insight into previous studies that have been carried out in the area of design and code defects and various solutions that have been proposed. Techniques that have been proposed by researchers to assist developers analyse and detect code defects are described to bring the researcher abreast of the state the art. In reviewing the literatures, gaps are identified and described.

Chapter three describes the methodology adopted in the research. It presents the framework of the thesis detailing steps taken to achieve the objectives stated in chapter one. The chapter also describes the steps taken to evaluate the implementation and data used to validate it.

The proposed concept is described in details in chapter four. The frameworks, architectural designs, algorithms and the details of the automated technique and its implementation are described. The chapter closes with screenshots of output generated by the prototype tool, PySTect to prove the concept enunciated in the technique.

Chapter five presents the results and discussions of the experiments conducted to answer research questions. The results of PySTect detections and validation by human subjects is presented. Others are results of experiments to find how density of code smells changes with evolution and what are the predominant smells in SDK domain. The results are discussed and conclusions drawn.

Chapter six describes benefits, limitations and suggestions for future work of the research. Lastly, a general conclusion of the entire research is drawn to provide a summary of what has been achieved by the research.

REFERENCES

Abbes, M., Khomh, F., Guéhéneuc, Y. G., & Antoniol, G. (2011). An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 181–190. https://doi.org/10.1109/CSMR.2011.24

Ahmed, I., Ghorashi, S., & Jensen, C. (2014). An Exploration of Code Quality in FOSS Projects. In *IFIP International Conf. on Open Source Systems*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-55128-4_26

Arcelli Fontana, F., & Zanoni, M. (2017). Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, *0*, 1–16. https://doi.org/10.1016/j.knosys.2017.04.014

Bassey, I., Dladlu, N., & Ele, B. (2016). Object-Oriented Code Metric-Based Refactoring Opportunities Identification Approaches: analysis. https://doi.org/10.1109/ACIT-CSII-BCD.2016.24

Bavota, G., De Lucia, A., Marcus, A., & Oliveto, R. (2013). *Using structural and semantic measures to improve software modularization*. *Empirical Software Engineering* (Vol. 18). https://doi.org/10.1007/s10664-012-9226-8

Bennett, K. H., & Rajlich, V. T. V. T. (2000). Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering (ICSE '00)* (Vol. 225, pp. 73–87). Limerick, Ireland: ACM New York, NY, USA. https://doi.org/10.1145/336512.336534

Bott, R. (2014). *Learning Python 3rd Ed. Igarss 2014*. https://doi.org/10.1007/s13398-014-0173-7.2

Bowes, D., Randall, D., & Hall, T. (2013). The Inconsistent Measurement of Message Chains, 62–68.

Brown, W. J., Malveau, R. C., Mowbray, T. J., & Wiley, J. (1998). *AntiPatterns: Refactoring Software , Architectures, and Projects in Crisis*. (T. M. F. Hudson, Ed.), *John Wiley & Sons, Inc* (Vol. 3). Robert Ipsen. Retrieved from http://www.amazon.com/dp/0849329949

Bulychev, P., & Minea, M. (2008). Duplicate code detection using anti-unification. In *In Proceedings of the 2nd Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE),*.

Chatzigeorgiou, A., & Manakos, A. (2014). Investigating the evolution of code smells in object-oriented systems. *Innovations in Systems and Software Engineering*, *10*(1), 3–18. https://doi.org/10.1007/s11334-013-0205-z

Chen, Z., Chen, L., Ma, W., & Xu, B. (2016). Detecting Code Smells in Python Programs. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. https://doi.org/10.1109/SATE.2016.10

Delchev, M., & Harun, M. F. (2015). Investigation of Code Smells in Different Software Domains. *Full-Scale Software Engineering FsSE*, (November), 31–36.

Dewes, A., & Neumann, C. (2018). *Python Anti-Patterns*. Berlin, Germany: QuantifiedCode. Retrieved from https://docs.quantifiedcode.com/python-anti-patterns/

Dos Reis, J. P. D., Brito E Abreu, F., & De Carneiro, F. G. (2017). Code smells incidence: Does it depend on the application domain? In *Proceedings - 2016 10th International Conference on the Quality of Information and Communications Technology, QUATIC 2016* (pp. 172–177). https://doi.org/10.1109/QUATIC.2016.044

Fan, H., & Mu, Y. (2014). A PERFORMANCE TESTING AND OPTIMIZATION TOOL, 24–27.

Fard, A. M., & Mesbah, A. (2013). JSNOSE: Detecting javascript code smells. In *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013* (pp. 116–125). https://doi.org/10.1109/SCAM.2013.6648192

Fenton, N. E., & Neil, M. (1998). Software Metrics : Successes , Failures and New Directions, 1–19.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., & Figueiredo, E. (2016). A Review-based Comparative Study of Bad Smell Detection Tools.

Fokaefs, M., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2012). Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software*, *85*(10), 2241–2260. https://doi.org/10.1016/j.jss.2012.04.013

Fontana, F. A., B, M. Z., & Zanoni, F. (2015). A Duplicated Code Refactoring Advisor. In L. et al. (Eds.) (Ed.) (pp. 3–14). Switzerland: Springer International Publishing Switzerland. https://doi.org/10.1007/978-3-319-18612-2

Fontana, F. A., Ferme, V., Marino, A., Walter, B., & Martenka, P. (2013). Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. In *IEEE International Conference on Software Maintenance, ICSM* (pp. 260–269). IEEE Computer Society. https://doi.org/10.1109/ICSM.2013.37

Fontana, F. A., Ferme, V., Zanoni, M., & Yamashita, A. (2015). Automatic metric thresholds derivation for code smell detection. *International Workshop on Emerging Trends in Software Metrics, WETSoM*, *2015-Augus*(October), 44–53. https://doi.org/10.1109/WETSoM.2015.14

Fontana, F. A., Mangiacavalli, M., Pochiero, D., & Zanoni, M. (2015). On experimenting refactoring tools to remove code smells. In *Scientific Workshop Proceedings of the XP2015 on - XP '15 workshops* (pp. 1–8). Helsinki, Finland: ACM. https://doi.org/10.1145/2764979.2764986

Fontana, F. A., Mäntylä, M. V, Zanoni, M., & Marino, A. (2016a). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 1143–1191. https://doi.org/10.1007/s10664-015-9378-4

Fontana, F. A., Mäntylä, M. V, Zanoni, M., & Marino, A. (2016b). Comparing and Experimenting Machine Learning Techniques for Code Smell Detection 1 Introduction. *Empirical Software Engineering*, *Volume 21*, Pages 1143-119.

Fontana, F. A., Mariani, E., Morniroli, A., Sormani, R., & Tonello, A. (2011). An experience report on using code smells detection tools. *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, (August 2017), 450–457. https://doi.org/10.1109/ICSTW.2011.12

Fontana, F. A., Zanoni, M., Marino, A., & Mäntylä, M. V. (2013). Code smell detection: Towards a machine learning-based approach. *IEEE International Conference on Software Maintenance, ICSM*, 396–399. https://doi.org/10.1109/ICSM.2013.56

Fowler, M. (1997). Refactoring : Improving the Design of Existing Code, 1–82.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*.

Ganesh, S. G., Sharma, T., & Suryanarayana, G. (2011). Towards a Principle-based Classification of Structural Design Smells, *12*(2), 1–29. https://doi.org/10.5381/jot.2013.12.2.a1

Garcia, J., Popescu, D., Edwards, G., & Medvidovic, N. (2009). Toward a Catalogue of Architectural Bad Smells. In C. Mirandola, R., Gorton, I., Hofmeister (Ed.), *Architectures for Adaptive Software Systems QoSA*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-02351-4_10

Griffith, I., Wahl, S., & Izurieta, C. (2011). TrueRefactor : An Automated Refactoring Tool to Improve Legacy System and Application Comprehensibility. In *In the Proceedings of the 24th International Conference of Computer Applications in Industry and Engineering (CAINE)*.

Khomh, F., Penta, M. Di, Guéhéneuc, Y. G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. In *Empirical Software Engineering* (Vol. 17, pp. 243–275). IEEE. https://doi.org/10.1007/s10664-011-9171-y

Lanza, M., & Marinescu, R. (2006). *Object-Oriented Metrics in Practice*. Springer-Verlag.

Lee, J., Lee, D., Kim, D., & Park, S. (2012). A Semantic-Based Approach for Detecting and Decomposing God Classes. In *ArXiv e-prints*. Retrieved from http://arxiv.org/abs/1204.1967

Lin, Y., & Holt, R. C. (2004). Formalizing Fact Extraction. *Electronic Notes in Theoretical Computer Science, 94*, 93–102. https://doi.org/10.1016/j.entcs.2004.01.001

Liu, X., & Zhang, C. (2017). DT : a detection tool to automatically detect code smell in software project, *71*(Icmmita 2016), 681–684.

Lott, S. F. (2014). *Mastering Object-oriented Python*. Birhingham - Mumbai: Packt Publishing Ltd.

Mansoor, U., Kessentini, M., Maxim, B. R., & Deb, K. (2016). Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal*, 1–24. https://doi.org/10.1007/s11219-016-9309-7

Mäntylä, M. V., & Lassenius, C. (2006). Subjective evaluation of software evolvability using code smells: An empirical study. In *Empirical Software Engineering* (Vol. 11, pp. 395–431). https://doi.org/10.1007/s10664-006-9002-8

Marinescu, C., Marinescu, R., Mihancea, P., Ratiu, D., & Wettel, R. (2005). iPlasma:An Integrated Platform for Quality Assessment of Object-Oriented Design. *Proceedings of the 21st IEEE International Conference on Software Maintenance ICSM 2005*, (August 2016), 77–80.

Marinescu, R, Ganea, G., & Verebi, I. (2010). InCode: Continuous Quality Assessment and Improvement. *2010 14th European Conference on Software Maintenance and Reengineering*, 274–275. https://doi.org/10.1109/CSMR.2010.44

Marinescu, Radu. (2004). Detection Strategies : Metrics-Based Rules for Detecting Design Flaws.

Mihancea, P. F., & Marinescu, R. (2005). Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems.

Moha, N., Duchein, L., Guehenuec, Y.-G., & Meur, L. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, *36*(1), 20–36. https://doi.org/10.1007/s00165-009-0115-x

Moha, N., Guehenuec, Y.-G., Meur, L., Laurence, D., & Tiberghien, A. (2010). From a Domain Analysis to the Specification and Detection of Code and Design Smells. *Formal Aspectes of Omputing*. https://doi.org/10.1007/s00165-009-0115-x

Moonen, L. M. F. (2002). Java Quality Assurance by Detecting Code Smell. *UvA-DARE (Digital Academic Repository) Exploring Software Systems*.

Munro, M. J. (2005). Product Metrics for Automatic Identification of " Bad Smell " Design Problems in Java Source-Code, (Metrics).

Murphy-Hill, E., & Black, A. P. (2010). An interactive ambient visualization for code smells. *Proceedings of the 5th International Symposium on Software Visualization*, 5–14. https://doi.org/10.1145/1879211.1879216

Murphy-Hill, E., Zimmermann, T., Bird, C., & Nagappan, N. (2014). The Design Space of Bug Fixes and How Developers Navigate It. *IEEE Transactions on Software Engineering*, 1–1. https://doi.org/10.1109/TSE.2014.2357438

Nunez-Varela, A. S., Perez-Gonzalez, H. G., Martinez-Perez, F. E., & Soubervielle-Montalvo, C. (2017). Source code metrics: A systematic mapping study. In *Journal of Systems and Software* (Vol. 128, pp. 164–197). Elsevier Inc. https://doi.org/10.1016/j.jss.2017.03.044

Olbrich, S. M., Cruzes, D. S., & Sjøberg, D. I. K. (2010). Are all Code Smells Harmful ? A Study of God Classes and Brain Classes in the Evolution of three Open Source Systems.

Ouni, A. (2014). *A Mono- and Multi-objective Approach for Recommending Software Refactoring*. Université de Montréal.

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., & Salah, M. (2015). The Journal of Systems and Software Improving multi-objective code-smells correction using development history. *The Journal of Systems & Software*, *105*, 18–39. https://doi.org/10.1016/j.jss.2015.03.040

Ouni, A., Kessentini, W., Member, S., & Kessentini, M. (2014). A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection, (September). https://doi.org/10.1109/TSE.2014.2331057

Padilha, J., Pereira, J., Figueiredo, E., Almeida, J., Garcia, A., & Sant'Anna, C. (2014). On the effectiveness of concern metrics to detect code smells: An empirical study. In J. M. et Al. (Ed.), *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 8484 LNCS, pp. 656–671). Springer, Cham. https://doi.org/10.1007/978-3-319-07881-6_44

Paiva, T., Damasceno, A., Padilha, J., Figueiredo, E., & Sant'Anna, C. (2015). Experimental Evaluation of Code Smell Detection Tools. *3th Workshop on Software Visualization, Evolution, and Maintenance (VEM 2015)*, 8.

Palomba, F. (2016). Alternative Sources of Information for Code Smell Detection : Postcards From Far Away. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Raleigh, NC, US: IEE.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., & De Lucia, A. (2015). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, *41*(5), 462–489. https://doi.org/10.1109/TSE.2014.2372760

Peters, T. (2010). The Zen of Python. In *Zen of Python* (pp. 301–302). Apress. Retrieved from http://download.springer.com/static/pdf/898/chp%253A10.1007%252F978-1-4302-2758-8_14.pdf?originUrl=http%3A%2F%2Flink.springer.com%2Fchapter%2F10.1007%2F978-1-4302-2758-8_14&token2=exp=1497511758~acl=%2Fstatic%2Fpdf%2F898%2Fchp%25253A10.1007%25252F978-1-43

Rasool, G., & Arshad, Z. (2015). A Review of Code Smell Mining Techniques. *Journal Software Evolution and Process*, 27:867–895. https://doi.org/10.1002/smr.1737

Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2016). A Large Scale Study of Multiple Programming Languages and Code Quality. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, *1*, 563–573. https://doi.org/10.1109/SANER.2016.112

Reitz, K., & Schlusser, T. (2016). *The Hitchhiker's Guide To Python* (1st Editio). Sebastopol, CA 95472: O'Reilly Media, Inc.

Schumacher, J., Zazworka, N., Shull, F., Seaman, C., & Shaw, M. (2010). Building empirical support for automated code smell detection. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10* (p. 1). Bolzano-Bozen, Italy.: ACM-IEEE International Symposium on Empirical Engineering & Measurement. https://doi.org/10.1145/1852786.1852797

Sjoberg, D. I. K., Yamashita, A., Anda, B. C. D., Mockus, A., & Dyba, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, *39*(8), 1144–1156. https://doi.org/10.1109/TSE.2012.89

Sz, G., Antal, G., Nagy, C., Ferenc, R., & Gyimóthy, T. (2017). The Journal of Systems and Software Empirical study on refactoring large-scale industrial systems and its effects on maintainability, *129*, 107–126. https://doi.org/10.1016/j.jss.2016.08.071

Tahmid, A., Nahar, N., & Sakib, K. (2016). Understanding the Evolution of Code Smells by Observing Code Smell Clusters, 8–11. https://doi.org/10.1109/saner.2016.45

Travassos, G., Shull, F., Fredericks, M., & Basili, V. R. (1999). Detecting Defects in Object-oriented Designs: Using Reading Techniques to Increase Software Quality. *SIGPLAN Not.*, *34*(10), 47–56. https://doi.org/10.1145/320385.320389

Tripathy, P., & Naik, K. (2015). *Software Evolution and Maintenance*. Hoboken, New Jersey & Canada: John Wiley & Sons, Inc. https://doi.org/10.1002/9781118964637

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M. Di, De Lucia, A., & Poshyvanyk, D. (2017). When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Transactions on Software Engineering*, *43*(11), 1063–1088. https://doi.org/10.1109/TSE.2017.2653105

Van Rossum, G., & Drake Jr., F. L. (2011). The Python Language Reference Manual. Retrieved from https://www.amazon.com/Python-Language-Reference-Manual/dp/1906966141

Wake, W. C. (2003). *Refactoring Workbook*. Addison-Wesley Professional. Retrieved from http://www.amazon.com/dp/0321109295

Walter, B., Matuszyk, B., & Fontana, F. A. (2015). Including structural factors into the metrics-based code smells detection. In *XP 2015 Workshops*. Helsinki, Finland: ACM. https://doi.org/http://dx.doi.org/10.1145/2764979.2764990

Wang, B., Chen, L., Ma, W., Chen, Z., & Xu, B. (2015). An empirical study on the impact of Python dynamic features on change-proneness. https://doi.org/10.18293/SEKE2015-097

Wentworth, P., Elkner, J., Downey, A. B., & Meyers, C. (2012). How to Think Like a Computer Scientist : Learning with Python 3 Documentation.

Williams, L., Ho, D., Heckman, S., & Authors, C. (2005). Software Metrics in Eclipse. Retrieved May 24, 2017, from http://realsearchgroup.org/SEMaterials/tutorials/metrics/

Wohlin, C., Runeson, P., Bost, M., Ohlsson, M. C., Regnell, B., & Wessl6n, A. (2000). *Experimentation in software engineering An Introduction*. (V. . Basili, Ed.). The Kluwer International Series in Software Engineering.

Yamashita, A., & Moonen, L. (2012). Do code smells reflect important maintainability aspects? *IEEE International Conference on Software Maintenance, ICSM*, 306–315. https://doi.org/10.1109/ICSM.2012.6405287

Yamashita, A., & Moonen, L. (2013a). Exploring the impact of inter-smell relations on software maintainability: An empirical study. *Proceedings - International Conference on Software Engineering*, 682–691. https://doi.org/10.1109/ICSE.2013.6606614

Yamashita, A., & Moonen, L. (2013b). To what extent can maintenance problems be predicted by code smell detection? -An empirical study. *Information and Software Technology*, *55*(12), 2223–2242. https://doi.org/10.1016/j.infsof.2013.08.002

Zaytsev, V., & Bagge, A. H. (2014). Parsing in a Broad Sense. In *17th International Conference, Models 2014, Valencia* (pp. 50–67). Valencia, Spain: Springer International Publishing Switzerland.

Zhang, M., Baddoo, N., Wernick, P., & Hall, T. (2011). Prioritising Refactoring using Code Bad Smells. In *2011 Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE Computer Society. https://doi.org/10.1109/ICSTW.2011.69