Designing and Developing a Machine Learning based Code Smell Detection Technique

A Thesis

submitted in fulfillment of the requirement

for the award of degree of

Doctor of Philosophy

Submitted By

Amandeep Kaur

(Roll No.: 951503008)

Under the guidance of

Dr. Sushma Jain

(Associate Professor, Computer Science and Engineering Department)

Thapar Institute of Engineering & Technology, Patiala

and

Dr. Shivani Goel

(Professor, Computer Science and Engineering Department)

Bennett University, Greater Noida

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY

Patiala - 147004, Punjab, India

January, 2021

I would like to dedicate this thesis to my loving parents, brothers, sisters, and friends, who make me whole.

“Everything that comes to us that belongs to us if we create the capacity to receive it.”

By Rabindernath Tagore

Certificate

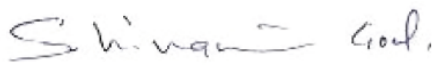
I hereby certify that the work which is presented in this thesis entitled "**Designing and Developing a Machine Learning based Code Smell Detection Technique**", in fulfillment of the requirement for the award of degree of "**Doctor of Philosophy**" submitted in Computer Science and Engineering Department of Thapar Institute of Engineering & Technology, Patiala, is an authentic record of my own work carried out under the supervision of **Dr. Sushma Jain** and **Dr. Shivani Goel** and refers other research works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

(Amandeep Kaur)
Regn. No. 951503008

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. Sushma Jain)
Associate Professor
Computer Science and Engineering Department
Thapar Institute of Engineering & Technology, Patiala, India



(Dr. Shivani Goel)
Professor
Department of Computer Science and Engineering
Bennett University, Greater Noida, India

Acknowledgement

Creating a PhD thesis is not an individual experience, rather it includes social platforms and several persons. Therefore, at this momentous occasion of completing my dissertation, I would like to acknowledge the contribution of all those benevolent people, I have been blessed to associate with. All the data collection, theories, algorithms would have failed to serve their purpose for me if their benedictions would not have joined hands with my efforts.

I shall begin with God the almighty. Without his will, I would have never found the right path. His mercy is with me throughout my life and even more in this study. I thank him for enlightening my soul with the respected love and compassion for the other humans and allowing me to enter a field where I could practice this desire. I proudly offer my gratitude towards god "**Waheguru ji**" for showering his divine blessings on me, for enlightening my life with his presence, for the good health and wellbeing, that were necessary to complete this thesis, for giving me the strength and patience to work through all these years so that today I can stand proud with my head held high.

My first and foremost vote of thanks go to the architect who shaped my dreams into the reality, my guide and mentor **Dr. Sushma Jain**, Associate Professor, Department of Computer Science, Thapar institute of Engineering & Technology, Patiala. She steered me through this journey with her valuable lessons, stimulating discussions and consistent encouragement. I have learnt extensively from her including how to raise new possibilities, how to regard an old question from a new perspective, how to approach a problem by systematic thinking, and exploiting serendipity.

Also, I feel privileged for having **Dr. Shivani Goel**, Professor, Department of Computer Science, Bennett University, Noida, who introduced me to the world of computer research and provided continuous support during my Ph.D. study and research. I acknowledge her for her patience, positive attitude, motivation, enthusiasm, and immense knowledge. I could not have imagined having a better

advisor and mentor for my Ph.D. study. She is a wonderful and generous advisor who has been through a lot and I admire her positive outlook and her ability to smile despite the situation. If I will stand proud of my achievements, then undeniably my both mentors are the main creditors. Every time I meet them, I can feel ***"fire in the belly"***. It is my privilege to be under their tutelage. They have taught me another aspect of life, that, ***"goodness can never be defied and good human beings can never be denied"***.

I am indebted to **Dr. Maninder Singh**, Professor and Head, CSED, Thapar Institute of Engineering & Technology, Patiala, for providing me the necessary administrative assistance and encouragement that made me explore innumerable things and also for providing necessary facilities that helped me in the completion of my research work.

I am sincerely grateful to my doctoral committee members: **Dr. Rinkle Rani**, Associate Professor, CSED, who stood by my side in the hard times, without her, it was impossible to reach at the present stage, **Dr. Vinay Arora**, Assistant Professor, CSED, and **Dr. MD Singh**, Associate Professor, EIED, Thapar Institute of Engineering & Technology, Patiala; for their helpful suggestions and regularly ensuring the progress of my research work. Also, I am thankful to the faculty, staff members, Librarians, and research scholars of CSED for their constant support.

I express my heartfelt gratitude to **Dr. S.S. Bhatia**, Dean Academics, Thapar Institute of Engineering & Technology, Patiala, for being with me as an elderly figure throughout. He was always there for me with his supporting hands whenever I needed it the most. He always taught me how a person can succeed in achieving what seems impossible to begin with.

My special vote of thanks also goes to **Dr. Parshant Rana**, Assistant Professor, CSED, Thapar Institute of Engineering & Technology, Patiala, for his perpetual guidance, stimulation and furnished support that has always kept me going ahead. I owe a lot of recognition to him for always being there for me. I feel privileged to

be associated with a person like him during my life.

I offer my deepest gratitude to my father, **S. Sarwan Singh** whose love and constant encouragement has been a major source of inspiration in turning my vision into reality. I express my heartful beholden to my mother **Mrs. Surinder Kaur**, I remember my mother's prayers and they have followed me. They have clung to me all the time as mothers provides the purest and unconditional love. I am strong and able to achieve this because a strong woman raised me.

I pay my deepest appreciation to my sister **Ms. Satnam Kaur** for helping me enormously, especially with mammoth task of proof reading of my thesis. We go through long hours' phone conversations at crucial times in which she helped me find the strength to continue my work here and guided me to come to the decision, once and for all that obtaining this degree would not be the first challenge in my life that I would not rise to meet. She is my teacher, my defense attorney, my personal press agent, even my shrink. She has selflessly encouraged me to explore new directions in life and seek my own destiny. I dedicate this milestone to her.

I am highly contented to thank my sister **Ms. Harmanpreet Kaur**, for paying heed towards me even in her own tough times and my loving brother **Mr. Lovepreet Singh** for being a golden thread to the meaning of my life. Siblings are like angels, with a love that glows forever. They are bringing the best out of me. My siblings have been my best friends all my life and I love them dearly and thank them for all their advice and support. I know I always have my family to count on when times are rough.

My special words of praise also go to my parental uncles **Late S. Gurpreet Singh** who always boosted me whenever I lose hope, **S. Kuldip Singh** for his continuous love and care specially for picking me up and dropping to bus stand whenever I used to come home. I also extend my thanksgiving to my parental aunts **Mrs. Kuldip Kaur, Mrs. Harmeet Kaur, Mrs. Rajwant Kaur, Mrs. Balwinder Kaur** and **Mrs. Palwinder Kaur** for the delicious food and blessings.

How can I forget to mention the angels of my life? **Sunnanatpreet Kaur, Gurswaroop Singh, Amolakpreet Kaur, Jashanpreet Kaur, Sehajpal Singh, Kamaljeet Singh, Harpreet Singh and Harpreet Kaur**. I tip my hat to the kids for relaxing me by playing with me and making my day wonderful with their beautiful smiles as smiling came out to be one of the best remedies. They taught me about having a good sense of humour and a good approach to life. I love you more than anything and I appreciate all your patience and support during my Ph.D. studies. I still remember how you keep on waiting for long hours to play with me and not disturbing me during my study hours. Although you have been very critical of my work, it was my pleasure to provide you with mountains of papers to play with and an over-heated laptop to sit on. I know one thing that ***"waiting is a sign of true love and affection"*** and you all proved that it's true.

Of course a big contentment to **Dr. Gaurav Dhiman**, for his eternal support and understanding of my goals and aspirations. His infallible love, care and support has always been my strength. His patience and sacrifice will remain my inspiration throughout my life. Without his help, I would not have been able to complete much of what I have done and become who I am. I have shared moments of deep anxiety but also of big excitement. His presence was very vital in a process that is often felt as tremendously solitaire. He always managed to make me feel special, appreciated my quirkiness and sense of humour, and has given me valuable support in the beginning of this study, the same way he did in the beginning of my career and has created the ease institutional conditions allowing me to stay here. He always had faith in me and my intellect even when I felt like digging hole and crawling into one because I didn't have faith in myself. These past several years have not been an easy ride, both academically and personally. I truly thank **Dr. Gaurav** for sticking by my side, even when I was irritable and depressed. Thanks for allowing me to vent my frustrations on you. And lastly by giving an endless help to finish the dissertation work.

A good support system is important to surviving and staying sane in graduation

school. I was lucky to be a part of one we like to call ***"tikkdi"*** and refer to **Dr. Gaurav Dhiman, Dr. Aman Sharma and myself**. These two friends formed the core of my research time. I couldn't have survived in the institute without them. We've all been there for one another and have taught ourselves and each other many tools and issues of programming and better article's writing. I know that I could always ask them for advice and opinions on lab related issues. I'll never forget wonderful lunches and fun activities we've done together.

I want to appreciate my friends and many years of friendship they have provided. I do not know how to extend my gratitude towards them. They were always there with me through the toughest moments, I still remember the time of final exams of our Ph.D. course work when I was crying very hard, unable to gather courage to study, at that time they taught me research methodology. I got good grades because they devoted the whole night in teaching me the subject. I acknowledge the time they've taken out of their busy schedule to help me out. Their friendly advice, soothing words and big heart helped me face all the obstacles and continue with my work. I will never forget their kindness. I must say ***"words cannot express my feelings, nor my thanks for all your help"***. Thank you!..**Dr. Sukhvir Kaur, Ms. Upasana Joshi, Ms. Latika, Dr. Manjit Kaur**.

With great appreciation I would like to acknowledge my other friends for their moral prop up and narration, which drives me to give my best. **Ms. Ankita wadhawan, Dr. Dilbag Singh** and **Ms. Prabhjot Kaur** for their sustenance. The time spent with them as PG mate was wonderful specially the evening tea and snacks time. I would also like to say a heartfelt thank you to **Mr. Sukhwinder Singh**, Graduation student and **Mr. Narpinder Singh**, Junior Research Fellow, **Dr. Sahil Vashist**, Ph.D. Scholar, Thapar Institute of Engineering & Technology, Patiala, for their assistance throughout my dissertation. The list is endless...thanks to one and all.

Hats Off to **Dr. Shikha Suheja**, who taught me how to make diagrams in short

time period. She always used to say ***"tum mouse nahi use karti na, issi liye tumhari Ph.D. late ho rahi hai"***. I miss the time spent with her at NITJ guest house where she made me dance on Sapna Chaudhry's songs at midnight, made me laugh out loud by cracking funny and weird jokes, made me enjoy the food and late night coffee. I really admire those times. I am tranquil to mention **Mr. Sandeep Verma** for being an underpin during my research years. He bolsters me up whenever needed by taking me to the movies, by making me party so that I can freshen up my mind, by improvising the grammatical errors. Thank you both for everything.

My deep appreciation goes out to **Dr. Pritpal Singh**, Vice-Chancellor, Sri Guru Granth Sahib University, Fatehgarh Sahib, for his kind support and who has been always helpful in numerous ways and was always ready to give their timely help whenever required.

I must thank **Dr. Navdeep Kaur**, Head, CSE, Sri Guru Granth Sahib World University, Fatehgarh Sahib, for her constant support, teachings and cooperation during this journey. I also gratefully acknowledge the faculty members **Dr. Kiran Preet kaur, Dr. Amandeep Kaur Virk, Ms. Harleen Kaur, Ms. Rupinder Kaur**, and **Ms. Manpreet Kaur** for their invaluable academics, personal assistance. Their friendly nature has always made me feel at ease with them and I could always look back on them for any support. Their timely help and friendship shall always be remembered. Thank you for bringing delicious homemade food for me and standing by side whenever required.

A special mention of thanks to technical staff of Sri Guru Granth Sahib World University **Mr. Karambir Singh, Mr. Ravinder Singh, Mr. Bhupinder Singh, Mr. Bableen Sodhi, Mr. Manpreet Singh, Mr. Dalwinder Singh, Mr. Shawinder Singh, Mr. Tajinder Singh Bawa, Mr. Maan Singh, Dr. Ravinderpal Singh, Mr. Maninder Singh Mehta** and **Ms. Amanpreet Kaur** for making golden memories in a chapter of my life. I truly appreciate all for helping me accomplish my goal.

I am indebted to **Ms. Baljinder Kaur, Mr. Jasbir Singh, Mr. Baljit Singh Batra, Ms. Sandeep Kaur Batra**, and their families in Patiala who opened their homes to me. They were always standing there as a strong pillar with me in the form of emotional strength. They always supported me like a family. It was like a second home here. They cooked food for me on my demands, celebrated my birthdays, made me a special part of their lives on special occasions and functions. I also extend my thanks to **Dr. Kuldeep Singh Batra** for strengthening and supporting me during the hardest phase of this study. Without his favour, it was very difficult to reach to the final stage of this journey.

My special regards to all my teachers specially **Mr. Amit Chhabra** and **Mr. Rajesh Mehta** because their teachings at different stages of education has made it possible for me to see this day. Because of their kindness I was able to reach a stage where I could write this thesis. In addition, I would like to mention **Dr. Navleen Kaur** and **Mrs. Deepika Sethi** for their wise counsel and sympathetic ear.

I also thank my students from the bottom of my heart **Mr. Akash Vanike, Mr. Vishal Arora, Ms. Nitika Arora, Mr. Ramandeep Singh, Mr. Sukhvir Singh**, and **Mr. Tejvir Singh Waraich**. They are all a fun bunch with lots of enthusiasm and optimism. They remind me how I used to be in their class. I haven't gotten a chance to get to really know them since I've been in a hole this past year but they've all been so friendly and personable to me. I wish these younger students the best of luck.

I'm also glad to have Ph.D. Scholars **Ms. Pawanjot Kaur**, **Ms. Pardeep Kaur**,**Mr. Kulwinder Singh**and **Mr. Jagpreet Singh**, as my friends. They never make me feel alone in the hostel and university campus. I value their friendship and support as well. My heart felt regard goes to **Dr. Karan Gumber**, my best friend, who encouraged me to further pursue with my studies by doing a Ph.D. in this field. This work was not possible without his constant support and scientific help. I do not know how to extend my gratitude towards him. He was always there with

me through the toughest moments. The more he was far in terms of distance, the more he was close to my heart. Instead of being in France he continuously supported me through social media.

I want to say that your commitment to excellence has inspired me a lot. I appreciate **Mr. Atinderpal Singh, Dr. Gaurav Jha** and **Mr. Gurjinder Singh Baath** for their innovative thinking. You are like a positive force and attitude around me and brought laurels to my life. Keep up the good work.

From the bottom of my heart I would like to say big thank you for showing me the ropes in my first months of this study, for always being so concerned about the state of my struggles with this PhD, for the discussions, for the sleepless nights we were working together before deadlines, for tolerating my constant emotional breakdowns and desire for silence, giving me shoulder to cry on when I need it specially during articles rejection, for sharing our good and bad experiences, from crying to laughing together and for all the fun we have had being roommates. Our many late night conversations about life, science, and people who have been very important to me. you will always have a special place in my heart.
Thank you for always being so sweet and caring and dropping everything to take breaks in the canteen with me when I needed to vent or collapse into tears. My memories with you are the ones I cherish the most. Thank you for putting up with the years of epic drama, and always being my partner in crime. It is a little bit disgusting how you can be so happy all the time, but it is a wonderful quality and I am grateful you shared that happiness with me. Thank you!...**Ms. Bindia Singla**. She is the best thing god has gifted me in this stage of life.

Also I thank to my other roommates and hostel friends **Ms. Anishu, Mrs. Aman, Ms. Shubhdeep Kaur, Ms. Kanchan Bala, Ms. Khushdeep Kaur, Ms. Priya** and Assistant Warden **Ms. Sarabhjit Kaur** who have always assisted me in the hostel duties at Bebe Nanki Girls Hostel of Sri Guru Granth Sahib World University, Fatehgarh Sahib. I will miss all the functions organised and attended here like

bangle ceremonies, hostel nights, Lohri celebration, Diwali Celebration and many more.

I am also thankful to my PG owners **Mr. Karnail Singh, Ms. Jaswinder Kaur**, their kids **Ms. Amritpal Kaur, Mr. Jatinder Singh** for giving me an equal place as a family member for years. I am also grateful to **Mr. Pardeep Singh, Ms Rama Rani, Mr. Sanjay Dadwal, Ms. Rano Dadwal** and little flowers of their family garden **Mr. Abhi, Mr. Ishu** and **Ms. Sanjana** for giving me an infinite love and affection.

I extend my deepest gratitude to the mess workers **Mr. Naresh Kumar, Mr. Sodhi, Mr. Imraan** and **Mr. Jha** for serving me hot food on time. Also, my very special thanks to sweepers **Ms. Baby Rani, Ms. Poonam, Ms. Sandeep** and **Ms. Shamsher Kaur**, for providing me clean and hygienic environment. My special regard go to all the security force of SGGS World University for providing safe environment to live in. Also, I am grateful to Cheema Diary for providing me pure milk on time. I want to show my contentment towards the owner of Ekam Pharmacy **Mr. Amardeep Singh Riar** for providing me medical assistance. My biggest thanks goes to **Tuffy** my pet for always cheering my mind and making me feel good in tensed situations.

As always it is impossible to mention everybody who had an impact to this work however there are those whose spiritual support is even more important. At last I also dedicate this thesis to my parents and all the persons who helped me directly or indirectly in completing my research degree.

Patiala (Amandeep Kaur)

March, 2020

Contents

Abstract

Software systems have become prevalent and significant in our today's society. These systems are becoming the core business of several industrial companies and, for this reason, these systems are getting bigger and more complex. In addition, these systems are subject to frantic modifications every day with respect to the introduction of new functionality or bug fixing operations. In this sense, developers also do not have the ability to design and execute ideal solutions, contributing to "code smells" being introduced.

Code smells refer to bad design and development practices commonly observed in software system. These smells reflect the sub-optimal design choices applied in the source code by developers. Code smells are the symptoms that indicate problems in the coding part of software which makes software hard to change and maintain. Several studies demonstrated the negative impact of code smells on the maintainability of software as well as on the ability of developers to comprehend a software system. That is why, several automated techniques and tools have been devised to discover parts of code affected by design flaws in order to improve their quality. Most of these techniques rely on the analysis of the structural properties (e.g., method calls) mined from the source code.

Despite the efforts of academicians and practitioners in recent years, there are still limitations that threaten the industrial applicability of techniques and tools for code smell identification. Specifically, there is a lack of evidence regarding the circumstances that lead to the introduction of code smells and the real effect of code smells on maintainability, since previous research focused the attention on a small number of software projects. Furthermore, in literature, the existing code smell detectors might be inadequate for detecting many code smells. One reason for inadequacy includes the dependence of existing techniques on only the structural properties of software systems. However, instead of structural properties extractable from the source code, a variety of code smells are intrinsically characterized by how code elements evolve over time.

There is a continual need of high quality software. Therefore, code smell detection and removal in the earlier phase will reduce the maintenance cost, helps the developers to improve software maintainability, readability and extendibility while increasing the speed at which programmer write their code and maintains the software. Code smell detection can be performed at various levels such as, requirement, design and coding. The primary focus of this thesis is on coding level, where software systems are improved using two steps: (a) detecting code smells and (b) prioritizing the code smells and analyzing their impact on software quality. To achieve these objectives, following three contributions are made in this thesis:

- First, J48 machine-learning algorithm is utilized for code smell detection. Code smell examples in the form of rules along with metrics specifications are given as a training dataset to J48. Code smell detection model is then trained and tested on considered dataset for finding code smells. The performance of proposed technique is evaluated on three Java open source softwares namely, GanttProject, Xerces-J, and Log4j to identify Blob, Functional Decomposition, Spaghetti Code, Feature Envy, and Data Class code smells.

  The results of J48 model are compared with some well-known machine-learning techniques and analyzed that the proposed model provides significantly better results. However, J48 suffers from the hyper-parameters tuning issue. Therefore, to tune the hyper-parameters of J48, a novel Sandpiper Optimization Algorithm (SPOA) is proposed. SPOA is further used in conjunction with J48 machine-learning algorithm called "SP-J48" to find the most significant set of metrics in order to identify code smells. SPOA is assessed on standard benchmark test functions to ensure its applicability in terms of convergence and computational complexity. The performance of the proposed algorithm is compared with other well-known optimization algorithms. Extensive experimental results indicate that the proposed SP-J48

provides significantly better results as compared to the competitive machine learning models.

Additionally, the proposed approach is extended by using C5.0 over J48 and named as "C5.0-SPOA" because C5.0 is a modification of J48 that provides accurate results, fast speed and generate smaller trees. C5.0-SPOA is then applied to identify eight code smells from five Java open source softwares. The performance of proposed approach is evaluated on the basis of Precision, Recall, and $F_{measure}$ performance metrics. The results show that the proposed approach provides significantly better results as compared to other existing techniques.

- Second, the proposed SPOA algorithm is further employed to prioritize the identified code smells for refactoring. The identified code smells are prioritized on the basis of three parameters such as, versioning history, architectural relevance, and code smell relevance. The performance of proposed prioritization approach is evaluated using three performance metrics namely, Code Smells Correction Ratio (CSCR), Estimated Effort (EE), and Severity of Fixed Code Smells of a system (SFCS). The experimental results reveal that the proposed prioritization approach helps the developers to reduce their efforts, saves time and improves productivity.

- Third, the impact of code smells prioritization on software internal quality attributes such as, cohesion, coupling, complexity, inheritance and size is analyzed. Three different software versions of each applications including, original version, version generated after removing code smells in a random order and version generated after removing code smells in prioritized sequence given by proposed prioritization approach are analyzed. Chidamber and Kemerer (C&K) metric suit is used to assess the impact of code smells on internal quality attributes as it is most commonly used and covers all aspects of internal quality measures. The obtained results show

that the code smells removed using prioritized sequence enhances the quality of software. Moreover, to validate these results, a pair-wise t-test is also conducted at 5% level of significance.

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

Abbreviation	Description
AMW	Average Methods Weight
AMWNAMM	Average Methods Weight of Not Accessor or Mutator Methods
ATLD	Access to Local Data
ATFD	Access to Foreign Data
AS	Antisingleton
BrC	Brain Class
BrM	Brain Method
BB	Blob
BCSA	BaseClassShouldBeAbstract
CBO	Coupling Between Objects Classes
CFNAMM	Called Foreign Not Accessor or Mutator Methods
CINT	Coupling Intensity
CYCLO	Cyclomatic Complexity
CLNAMM	Called Local Not Accessor or Mutator Methods
CMSC	Common Methods in Sibling Classes
Cmnt	Comment
CoC	Controller Class
CC	Changing Classes
CM	Changing Methods
CBSP	ClassDataShouldBePrivate

Abbreviation	Description
DupC	Duplicate code/code clone
DC	Data Class
DaC	Data Clump
Divc	Divergent Change
DIT	Depth of Inheritance Tree
EC	Empty Catch
FD	Functional Decomposition
FE	Feature Envy
FDP	Foreign Data Providers
GC	God Class
II	Inappropriate Intimacy
ILC	Incomplete Library Class
IMP	Usage of implementation instead of interface
ISPV	Interface Segregation Principle Violation
LOC	Lines of Code
LOCNAMM	Lines of Code Without Accessor or Mutator Methods
LzyC	Lazy Class
LC	Large Class
LM	Long Method
LPL	LongParameterList
LAA	Locality of Attribute Accesses
MM	Middleman
MspC	Misplaced Class
MC	Message Chain
MaMCL	Maximum Message Chain Length
MeMCL	Mean Message Chain Length
MAXNESTING	Maximum Nesting Level

Abbreviation	Description
NFL	No Free Lunch
NOPK	Number of Packages
NOCS	Number of Classes
NOM	Number of Methods
NOMNAMM	Number of Not Accessor or Mutator Methods
NOA	Number of Attributes
NOI	Number of Interfaces
NOC	Number of Children
NMO	Number of Methods Overridden
NIM	Number of Inherited Methods
NOII	Number of Implemented Interfaces
NODA	Number of default Attributes
NOPVA	Number of Private Attributes
NOPRA	Number of Protected Attributes
NOFA	Number of Final Attributes
NOFSA	Number of Final and Static Attributes
NOFNSA	Number of Final and non - Static Attributes
NONFNSA	Number of not Final and non - Static Attributes
NOSA	Number of Static Attributes
NONFSA	Number of non - Final and Static Attributes
NOABM	Number of Abstract Methods
NOCM	Number of Constructor Methods
NONCM	Number of non - Constructor Methods
NOFM	Number of Final Methods
NOFNSM	Number of Final and non - Static Methods
NOFSM	Number of Final and Static Methods
NONFNABM	Number of non - final and non-abstract Methods

Abbreviation	Description
NONFNSM	Number of Final and non - Static Methods
NONFSM	Number of non - Final and Static Methods
NODM	Number of default Methods
NOPM	Number of Private Methods
NOPRM	Number of Protected Methods
NOPLM	Number of Public Methods
NONAM	Number of non - Accessors Methods
NOSM	Number of Static Methods
NMCS	Number of Message Chain Statements
NOAM	Number of Accessor
NOPA	Number of Public Attribute
NOP	Number of Parameters
NOAV	Number of Accessed Variables
NOLV	Number of Local Variable
PSO	Particle Swarm Optimization
PF	Public Field
PIH	Parallel Inheritance Hierarchy
RFC	Response for a Class
RB	Refused Bequest
SSC	Switch Statement Case
SS	Shotgun Surgery
SAK	Swiss Army Knife
SG	Speculative Generality
SC	Spaghetti Code
Tca	Type Casting
TF	Temporary Field
TMP	Temporary variable used for several purposes

Abbreviation	Description
UP	Unused Parameter
WMC	Weighted Methods Count
WMCNAMM	Weighted Methods Count of Not Accessor or Mutator Methods

Chapter 1

Introduction

"Although this may seem a paradox, all exact science is dominated by the concept of approximation." By Bertrand Shaw

This chapter discusses the fundamental concepts of software maintenance, code smells taxonomy, code smell detection and prioritization, application of machine-learning techniques in code smell detection, decision tree algorithm, performance metrics, C5.0 algorithm, problem statement, objectives, methodology used, contribution to thesis, and outline of the thesis.

1.1 Software Maintenance

Software maintenance is defined as an activity performed for modification of software product after its release to correct faults or other attributes, to improve its performance, or to keep it useable in changing environment [1, 2, 3]. There are four types of maintenance activities which are defined as:

- *Adaptive Maintenance:* It is concerned with the modifications made to a software after its delivery to make it adaptive to a new environment, i.e., to run on a new operating system.
- *Corrective Maintenance:* It is concerned with the changes made to a software product after its public release to correct its faults which may occur due to error in design, coding or logic.
- *Perfective Maintenance:* It is defined as modifications made to a software

product while adding new functionalities to improve its performance.

- *Preventive Maintenance:* It deals with preventing an error before it actually occurs.

These maintenance tasks were firstly defined by Swanson et. al [4] in 1976. The meaning of preventive maintenance is unambiguous and not even clear to software community. This issue was discussed in workshop entitled "do we know what preventive maintenance is?" [5].

There are some software definitions of software maintenance given by other researchers [6]. According to Haikala and Marijarvi [7], software maintenance is defined as fixing bugs, solving client's problems, changing program's functionality, adding new features while requirement of a customer changes. This definition is argued and contradicted by Glass and Noiseux [8]. According to them, changes made to a software product after its delivery is maintenance. This statement is also supported by Pigoski [9], who stated that all maintenance tasks are done in the post delivery stage of a software. The type of maintenance activities along with corresponding time consumed by each is shown in Fig. 1.1. Software maintenance

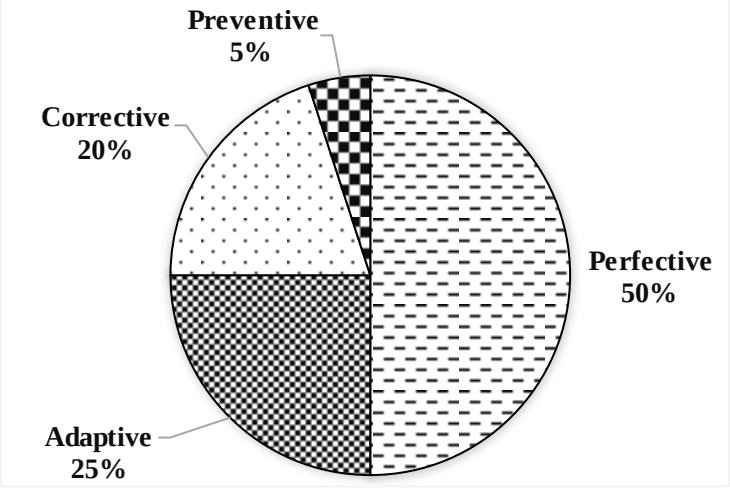


Fig. 1.1. Time consumed by each maintenance task.

is the last phase of software development life cycle as shown in Fig. 1.2. It comes

into use after the delivery of product and is considered to be most expensive phase.

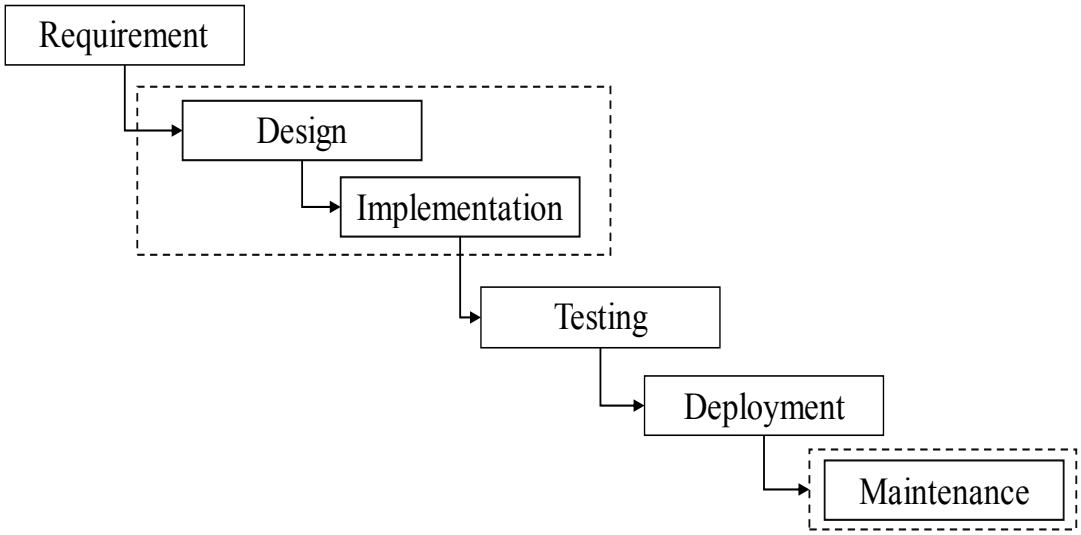


Fig. 1.2. Software development life cycle.

1.2 Measuring Maintainability

"You cannot control what you cannot measure" [10]. Therefore, like other quality attributes, software maintainability also needs to be measured. Measuring the size of a software confers great difficulty due to the involvement of various aspects such as reusability, functionality, redundancy, effort, and complexity [11]. For example: even if two different systems exhibit same features, different programmers can elucidate different difficulty levels. If a programmer have more experience it is likely that the program created by that developer is smaller in respect of complexity, redundancy and effort, but greater in respect of functionality and reusability.

This section focus on metrics that can be directly calculated from source code of a software and are called as source code metrics. These calculated values assist the researchers to draw reliable measurements of quality attributes based on these software metrics. Most of the maintainability studies focus on source code metrics for measuring maintainability because, these are easy to calculate,

most recognised and utilized by the researchers in the literature [12, 13]. There are generally two types of source code metrics such as Traditional metrics, and Object-Oriented metrics.

- Traditional metrics: In an Object-Oriented system, traditional metrics generally determine software complexity or the size of methods that comprise the operations of a class. The most commonly used traditional metrics include Lines of Code (LOC) [14] (abbreviated as NLOC and NCLOC), Halstead metrics [15], Comment Percentage (CP)[16], and McCabe's Cyclomatic Complexity [17].

- Object-Oriented metrics: Object-Oriented metrics quantify software quality by measuring the class or design properties of an Object-Oriented software. Numerous Object-Oriented metrics have been developed in the literature. Among them most popular metrics suits cited in the literature [18, 19] are Chidamber and Kemerer metrics (also called C&K metrics) [20], Li and Henry metrics [21], Metrics for Object-Oriented Design (MOOD metrics) [22], Lorenz and Kidd metrics [14], Quality Model for Object-Oriented Design metrics (QMOOD) [23], Metrics for Object-oriented System Environments (MOOSE metrics) [24, 20], and Extended Metrics for Object-Oriented Software Engineering (EMOOSE metrics) [25], etc. The hierarchy of source code metrics is shown in Fig. 1.3.

The concept of code smells is introduced in Object-Oriented software systems. Therefore, in this thesis, Object-Oriented metrics have been used.

1.3 Code Smell

Code smells are symptoms that indicate problems in software code which makes software tough to change and maintain [26]. Code smells are not the errors in code i.e., they do not prevent the software from working. Despite, these are weaknesses in the design part of a software system that can either slow down

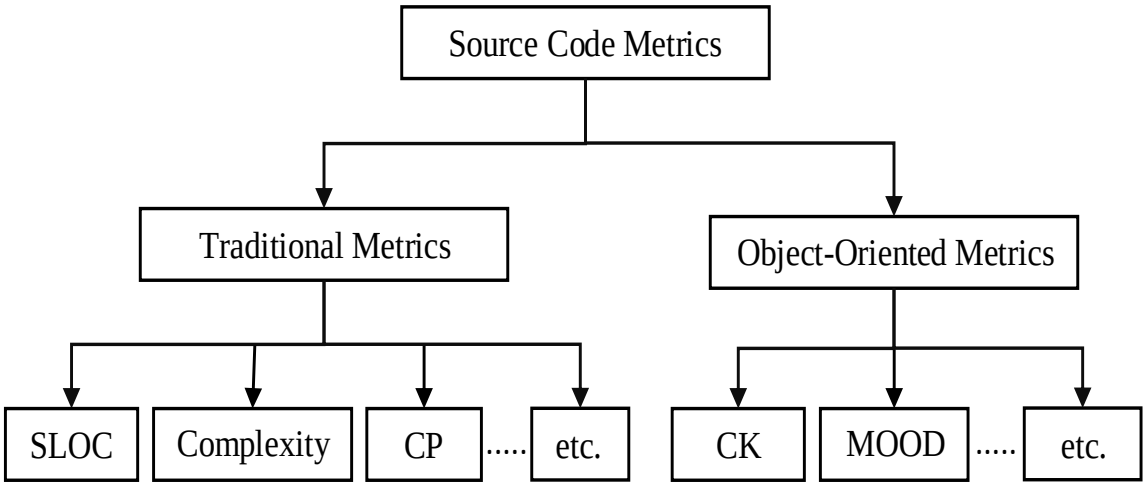


Fig. 1.3. Metrics hierarchy.

the development or increase the threat of failure or errors in future.

1.3.1 Evolution of Code Smell

A quality object-oriented software possesses various non-functional features like usability, reusability, changeability, and ease of evaluation. A common set of design principles namely data abstraction, modularity, and data encapsulation are followed by software systems during development to preserve these non-functional features [27]. But during maintenance, software undergoes several changes like addition of new requirements, quick bug fixes, or adaption to new environments. Due to a market rivalry, work deadline pressure or developers' inexperience, these changes are made to the software without considering the aforementioned design principles. The defilement of these design principles is one of the main sourcing factor behind the introduction of "code smells" [28, 29].

The presence of these code smells can also increase the maintenance cost, efforts, fault-proneness, defect-proneness, change-proneness and decrease the comprehensibility and understandability of a project. In some cases, code smells are introduced in the code by the developers while implementing vital patches or creating suboptimal choices in a hurry [30]. The group of code smells is sometimes known as anti-patterns. The term anti-pattern was coined by Opdyke in 1992 [31]. Anti-patterns are the sign of existence of code smells in source code. There is a

small gap between code smells and anti-patterns. Code smells are also known as bad smells, code bad smells, anomalies, or design defects [11, 29, 31, 32, 33]. Henceforward, in this thesis, these terms are used interchangeably.

1.3.2 Need of Code Smell Detection

From last few years, bad smells have become a sign of software systems that can cause complications in maintaining software quality during maintenance. It is widely believed that code smells are one of the major threats to the quality of the software. Various researchers stated that the software maintenance tasks consume more than 70% of the whole cost of a software project [34, 9, 35]. To avoid the bad-practices and increase software maintainability, adaptability, understandability, and extensibility, this high cost could be significantly reduced by providing automatic or semiautomatic solutions, which assists the development team to remove smells as early as possible. This will further help in improving the efficiency and effectiveness of a software organisation by improving the quality of a software.

Code smells are bad programming practices that severely affect the development of a software project. It is believed that bad smells can cause failure indirectly, if not directly [11]. Overall, code smells make a system hard to maintain and may introduce bugs frequently. For example, Feature Envy code smell (see Fig. 1.4) where a method makes extensive usage of other class rather than the class in which it is currently located. The changes made in one class exploits the functionality of a system. In this example, the function phd() is repeatedly calling the methods from class FindTopic. Therefore, this method is Feature Envy code smell as it is more interested in the methods of another class than the one in which it is defined. This introduces additional coupling in the code.

A good quality software system requires low coupling and high cohesion. Thus, it should be avoided in order to increase the quality of software system. The best technique to remove the code smells is refactoring. Refactoring is defined as

```
Boolean phd(FindTopic findTopic)
{
   Student waitStudent= shell.getDisplay().getSystemStudent(SWT.STUDENT_WAIT);
   Shell.setStudent(waitStudent);
   Boolean meetCase = findTopic.getMeetCase();
   Boolean meetWord = findTopic.getMeetWord();
   String findString = findTopic.getFindString();
   Int column = findTopic.getChoosenFindArea();
}
```

Fig. 1.4. Example of feature envy code smell.

the change made to the internal structure of software system without changing its external behavior. Therefore, detecting and prioritizing these code smells for refactoring helps the developers to easily maintain, evolve and understand their software project [29]. Some of the prominant reasons of code smell occurrences are:

- Developers constraint and time limitations: The program is sporadically written in ideal environmental settings. The maintenance phase results in violation of design principles due to confinement of programmer's skills and rigid time restrictions [36].
- Complexity of a system: The complexity of a system increases with its growth, unless the effort is made to maintain or decrease it. The difficulty in understanding the developed software by new programmer results in orientation of code smells.
- Continuing change: Systems must be reformed repeatedly else their performance will degrade.
- Programming language Limitation: Due to limitation in programming knowledge, developers are forced to copy the code [36]. During development phase, they do not care about bugs introduced in the system

which later on results in introduction of code smells.

- Phobia of writing new code: Developers avoid to bring new ideas in the existing software as this practice can make the programme more complicated and may result in the introduction of more errors. So they make experiments with the existing code which violate the design principles [37, 38].

- Refactoring delay: Due to time limitation, programmers delay refactoring or restructuring of a software project [39], which subsequently leads to increase in maintenance cost and introduction of bugs.

1.3.3 Code Smell Taxonomy

The term code smell was coined by Fowler and Beck in 1999. They proposed 22 code smells in a single flat list but did not classify them into categories [29]. In 2003, Mantyla et al. [40] categorised Fowler's smells into seven groups according to their characteristics for better understanding and evaluation. In each type of code smell, a definite type of system components such as classes or methods is reviewed which can be evaluated from its characteristic [41]. These categories are 'Bloaters', 'Couplers', 'Object Orientation Abusers', 'Encapsulators', 'Change Preventers', 'Dispensables', and others. Code smells which do not fall into any of the six categories are placed in the other category. The diagrammatic description of this taxonomy is given in Fig. 1.5.

- Bloaters: Bloaters are classes, code or methods that have grown so large that they cannot be managed adequately. Code smells that fall under this category are 'Data Clumps', 'Long Method', 'Primitive Obsession', 'Large Class', and 'LongParameterList' [29, 40, 42, 43].

  - *Data Clumps:* Data Clumps are existence of identical set of variables at multiple places in the code. This code smell can be spotted by constantly looking for the similar data items (variables or parameters) passed around together. The example of data clump code smell in a

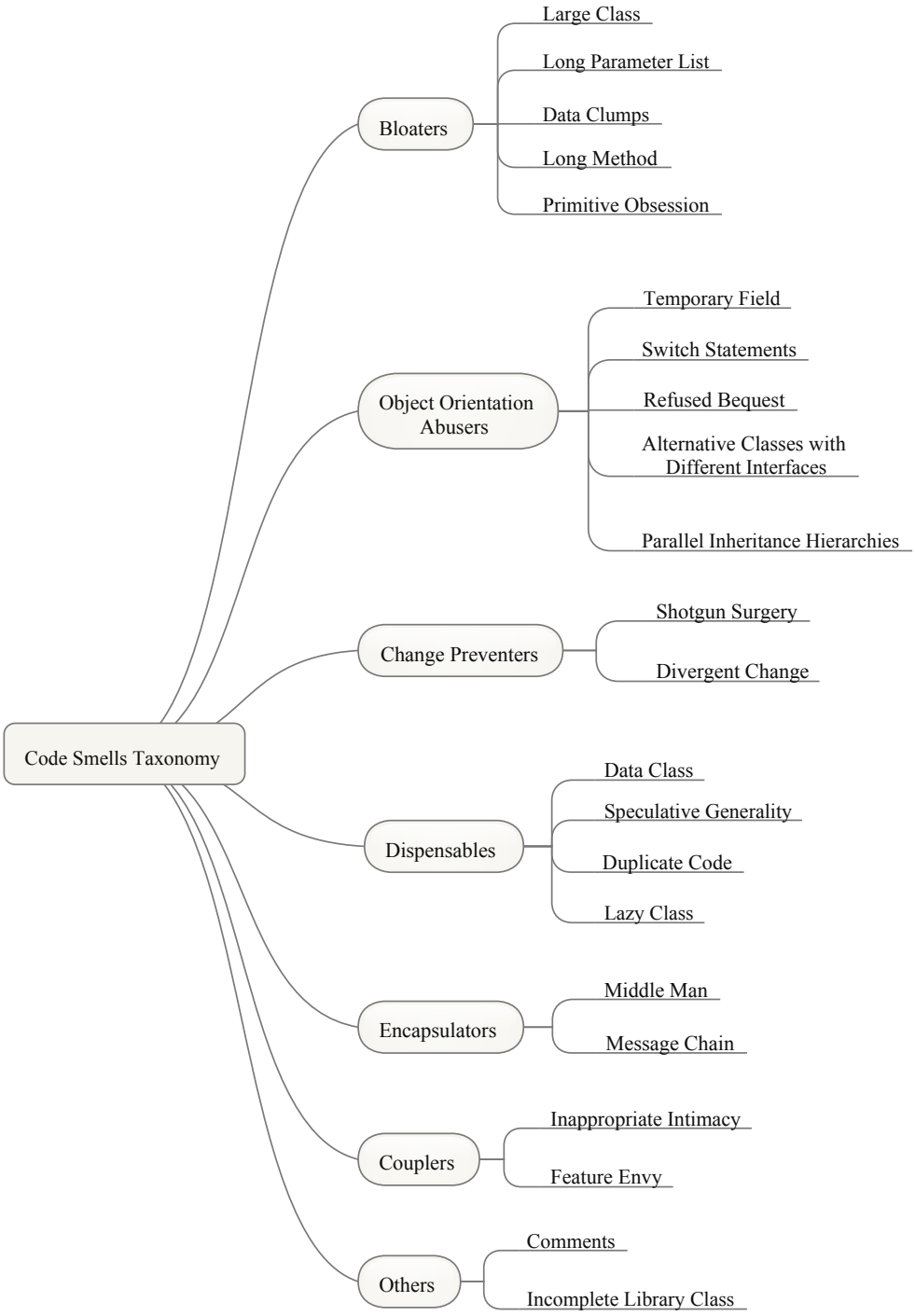


Fig. 1.5. Code smell taxonomy.

code part is shown in Fig. 1.6.

– *Long Method:* These are the methods that accomplish multiple actions and are comprised of multiple lines of code. Any method that contains greater than 15 lines of program is considered as a Long Method. Modifying or understanding a single 'Long Method' is more difficult

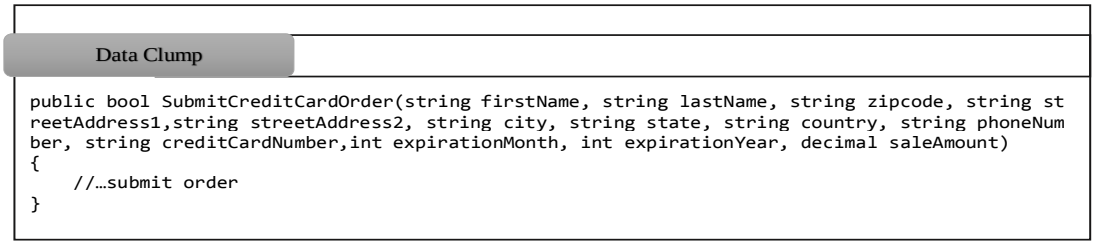


Fig. 1.6. Example of Data Clump code smell.

than understanding numerous smaller methods. The coding example of long method code smell is shown in Fig. 1.7.

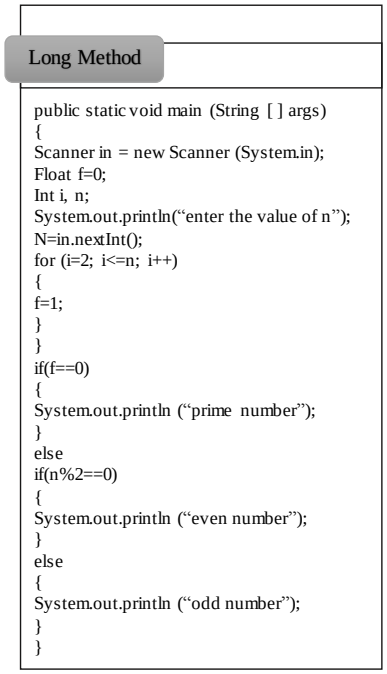


Fig. 1.7. Example of Long Method code smell.

– *Primitive Obsession:* It is not precisely a code smell, instead it signifies a symptom of the bad smell, and is building block of data in code. In order to perform simple tasks, the use of primitives instead of small objects represent the sign of this code smell. Example of Primitive Obsession is given in Fig. 1.8.

– *Large Class:* A class that encompass too many lines of code, methods or fields and has many duties is called a Large Class. This type of code

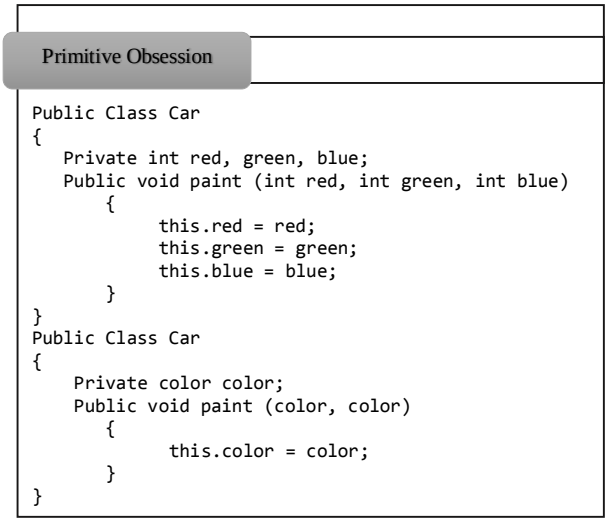


Fig. 1.8. Example of Primitive Obsession code smell.

smell is difficult to recite, recognize and troubleshoot. It is also called as God Class or Blob and its example is given in Fig. 1.9.

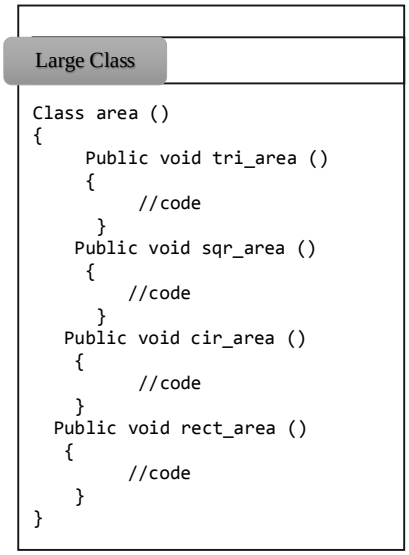


Fig. 1.9. Example of Large Class code smell.

– *LongParameterList:* When a method is passed more than three parameters, it is called as LongParameterList. A method with great number of parameters is more complex. It is better to limit number of parameters passed because LongParameterLists are inconsistent, difficult to learn and tough to use. Example of LongParameterList is shown in Fig. 1.10.

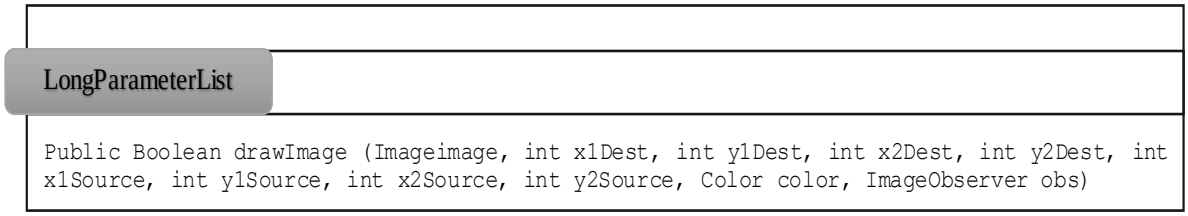


Fig. 1.10. Example of LongParameterList code smell.

- Object Orientation Abusers: This category is associated with circumstances where system does not exploit spontaneous and noticeable features of object oriented design. The reason behind this problem is prior experience of programmer in procedural languages and lack of knowledge and training about Object Oriented programming. Code smells under this category are 'Parallel Inheritance Hierarchies', 'Temporary Field', 'Alternative Classes with Different Interfaces', 'Switch Statement', and 'Refused Bequest'.

  – *Parallel Inheritance Hierarchies:* It is an extraordinary case of Shotgun Surgery which refers to duplicate class hierarchy. It occurs when we create a subclass for a class. This code smell can be identified by examining the identical prefixing of class names in different hierarchies. Example of this code smell is shown in Fig. 1.11.

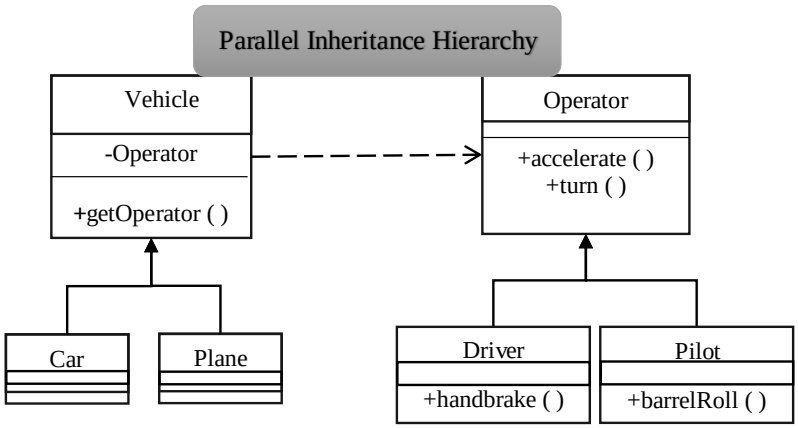


Fig. 1.11. Example of Parallel Inheritance Hierarchies code smell.

  – *Temporary Field:* This code smell occurs in those cases where the scope of a variable lies within the class instead of being in a method. This bad

smell disrupts the principle of information hiding. The code containing Temporary Field code smell is shown in Fig. 1.12.

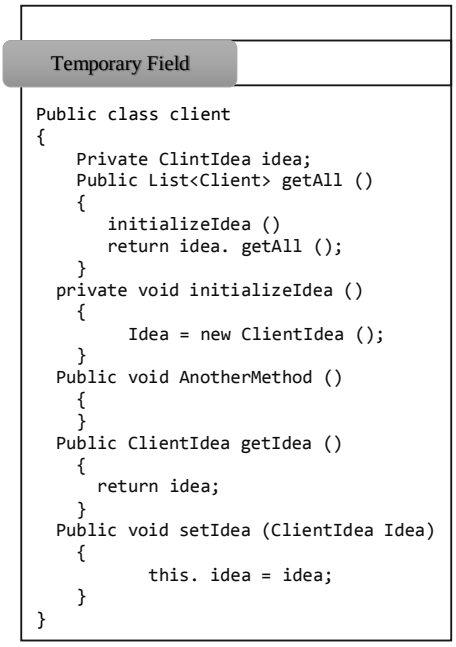


Fig. 1.12. Example of Temporary Field code smell.

– *Alternative Classes with Different Interfaces:* It arises when two methods with different signatures perform similar function in different classes. Fig. 1.13 displays the code with Alternative Classes with Different Interfaces code smell.

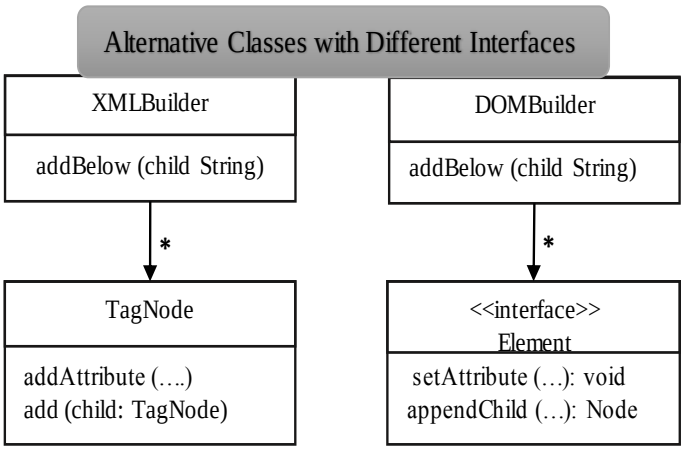


Fig. 1.13. Example of Alternative Classes with Different Interfaces code smell.

– *Switch Statement:* This code smell occurs when a sequence of

'if' statements are dispersed throughout the code or similar switch statements occur in a program. Fig. 1.14 shows the presence of switch statement code smell.

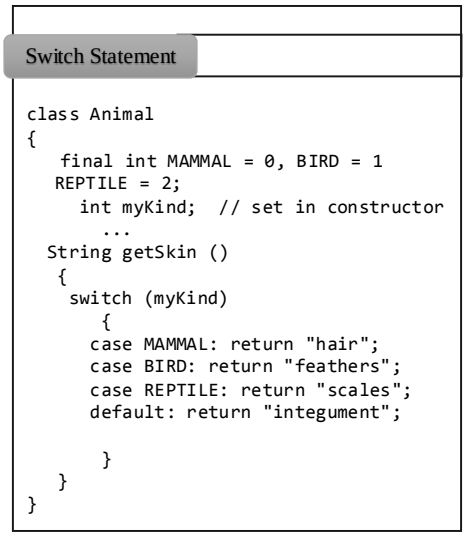


Fig. 1.14. Example of Switch Statement code smell.

– *Refused Bequest:* When a subclass inherited data and methods from parent class but does not need all that behaviour and it refuses some behaviour of parent class then 'Refused Bequest' code smell occurs. This kind of code smell is shown in Fig. 1.15.

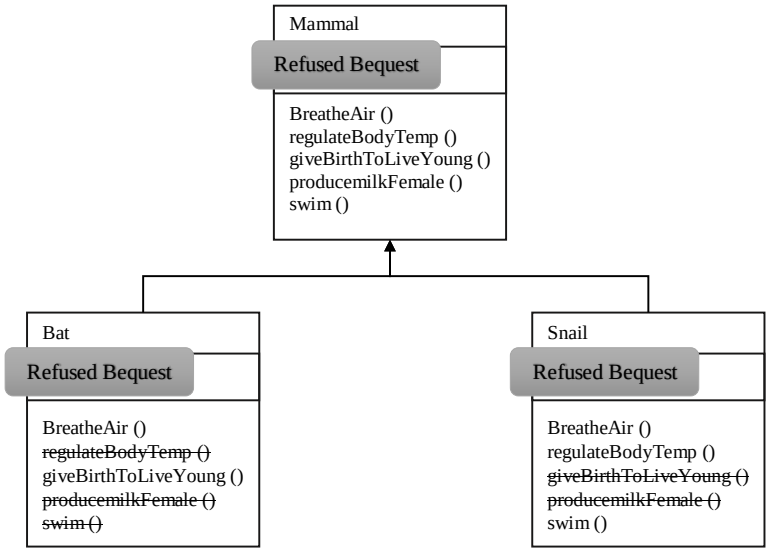


Fig. 1.15. Example of Refused Bequest code smell.

- Change Preventers: This category involves those code smells which hinder the software modification process. Smells under this group are 'Shotgun Surgery' and 'Divergent Change'.

  – *Shotgun Surgery:* In this code smell, a single modification made to a class leads to the alteration of many classes concurrently. An example of this kind of code smell is shown in Fig. 1.16.

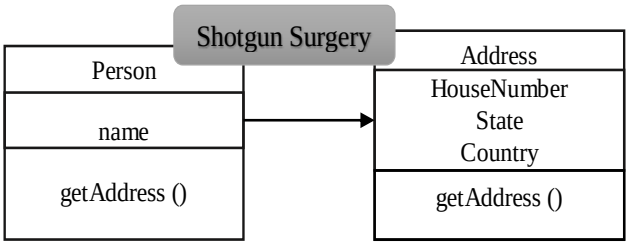


Fig. 1.16. Example of Shotgun Surgery code smell.

  – *Divergent Change:* This smell is opposite of Shotgun Surgery in the sense that it occurs when multiple changes are made to single class for different reasons whereas in Shotgun Surgery single alteration is made to several classes concurrently. Code with Divergent Change code smell is shown in Fig. 1.17.

- Dispensables: This group constitutes those code smells that contain something unnecessary and useless and whose removal can make code easier to understand, effective and cleaner. Code smells under this category are 'Data Class', 'Speculative Generality', 'Lazy Class', and 'Duplicate Code'.

  – *Data Class:* These classes are dumb data holder and encompass only public field's getter and setters methods. As these classes contain only data so they are used by other classes to perform their functionalities. Example of Data Class code smell is given in Fig. 1.18.

  – *Lazy Class:* It is a class that does not contain any responsibility and stays in the system for future use. It increases complexity of the system. We

Divergent Change

```
Public class Account
{
       Private int accountNumber;
       private double balance = 0;
    Public Account (int accountNumber)
        {
              this. accountNumber = accountNumber;
        }
    Public int getAccountNumber ()
        {
              Return accountNumber;
        }
    Public double getBalance ()
        {
              Return balance ();
        }
    Public void deposit (double amount)
        {
              Balance += amount;
        }
    Public void withdraw (double amount)
        {
              Balance -= amount;
        }
    Public String toXml ()
     {
       return "<account><id>" + integer. To String (getAccountNumber ()) + "</id>
       +"<balance>" + double. toString (getBalance ()) + "< / balance> </account>";
     }
}
```

Fig. 1.17. Example of Divergent Change code smell.

should either remove these classes or increase their responsibility. Fig. 1.19 shows the example of Lazy Class code smell.

– *Speculative Generality:* This type of code smell occurs when a class is developed with all sort of special cases and hooks. This results in harder maintainability and understanding of code. It can be identified when test cases are users of a method or a class. Code containing Speculative Generality code smell is shown in Fig. 1.20.

– *Duplicate Code:* This code smell results from the presence of similar code in two or more unrelated classes. It occurs when the identical expression is found in two different methods of similar class or in two familial subclasses. Fig. 1.21 shows the example of Duplicate Code code smell.

- Encapsulators: Code smells that deal with 'encapsulation' or 'data communication' mechanism are included in this group. This category

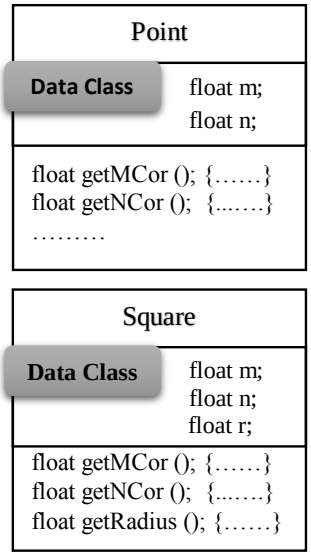


Fig. 1.18. Example of Data Class code smell.

includes 'Message Chain' and 'Middle Man' code smells.

- *Message Chain:* This code smell occurs from the chain of more than two user defined objects. For example, when a client class makes a call to object of another class which in turn calls the object of another third class and so on. Example of Message Chain is shown in Fig. 1.22.

- *Middle Man:* It is a class that assigns many of its task to classes which subsequently follow it. If there is excessive delegation, program can be delayed. Each time a delegant is needed to be added or modified while creating a new method or modifying older one. This type of code smell is shown in Fig. 1.23

- Couplers: This category contains the smells that occur due to excessive and harmful coupling in the code. 'Feature Envy' and 'Inappropiate Intimacy' are two smells related to coupling issues.

  - *Feature Envy:* It is a signal that a method makes extensive usage of other class rather than the class in which it is currently located. Fig. 1.24 shows the example of Feature Envy code smell.

  - *Inappropriate Intimacy:* When there is tight coupling between two

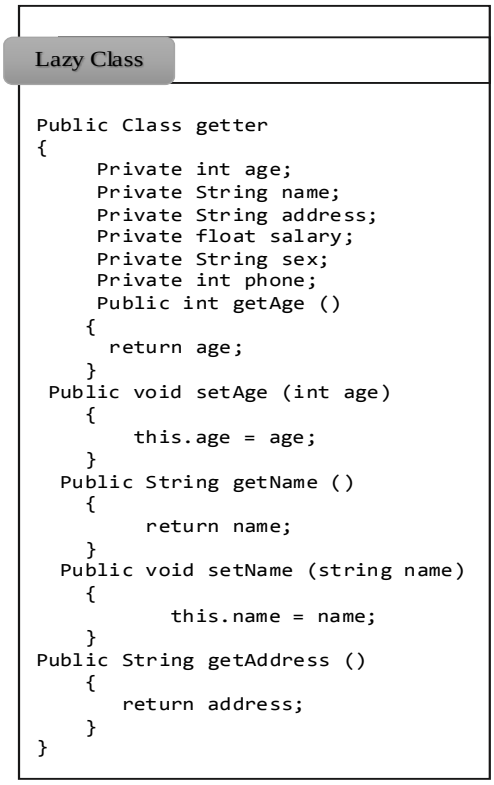


Fig. 1.19. Example of Lazy Class code smell.

classes and a huge time is spent by them in accessing each other's private variables then it is called 'Inappropriate Intimacy'. Fig. 1.25 shows the code containing Inappropriate Intimacy code smell.

- Others: The code smells which do not fit in any of the aforementioned six categories are included in this group. 'Comments', 'Incomplete Library Class' fall under this category.

  – *Comments:* These are sweet smells. These are used when we are not sure about what to do and why to do? If used excessively in code these are considered as code smells. Fig. 1.26 shows the example of comments code smell.

  – *Incomplete Library Class:* When more or lesser amount of functionality is exhibited by a library class than the required one then it is called as Incomplete Library Class. An example of this type of code smell is shown in Fig. 1.27.

Speculative Generality

```
Public class customer
{
    Private String name;
    Private String address;
    Private String gender;
    Private String country;
    Private MailingAddress mailingAddress;
    Public Customer {String name, String gender, String add, String country, MailingAddress mAdd)
    {
       this.name = name;
       this. address = add;
       this. gender = gender;
       this. country = country;
       this. mailingAddress=mAdd;
    }
  String getName ()
    {
          return name;
    }
  MailingAddress getMaailingAddress ()
    {
          return MailingAddress;
    }
}
```

Fig. 1.20. Example of Speculative Generality code smell.

Later, in 2006, Mantyla et al. [44] proposed a revised version of this taxonomy. Wake et al. [45] also presented a taxonomy of code smells. Code smells can also be considered as symptom of a design level flaw. Brown et. al presented these design level flaws in his workbook [33] and named them as Anti-Patterns. The authors defined a list of 40 Anti-Patterns into three categories namely development Anti-Patterns, architecture Anti-Patterns, and management Anti-Patterns. These Anti-Patterns include Functional Decomposition, Spaghetti Code, Walking through a Minefield, Continuous Obsolescence, Lava Flow, Ambiguous Viewpoint, Poltergeists, Mushroom Management, Dead End, Golden Hammer, Input Kludge, Cut-and-Paste Programming, Boat Anchor, Walking through a Minefield, Swiss Army Knife, Reinvent the Wheel, The Grand Old Duke of York, Autogenerated Stovepipe, Stovepipe Enterprise, Jumble, Stovepipe System, Cover Your Assets, Vendor Lock-In, Wolf Ticket, Warm Bodies, Design by Committee, Blowhard Jamboree, Analysis Paralysis, Viewgraph Engineering, Death by Planning, Fear of Success, Corncob, Intellectual Violence, Irrational

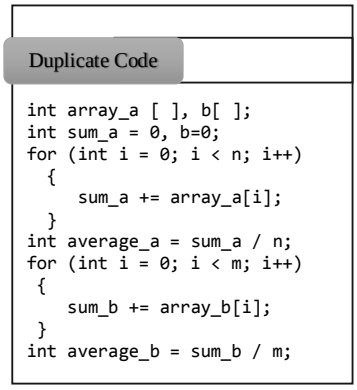


Fig. 1.21. Example of Duplicate Code code smell.

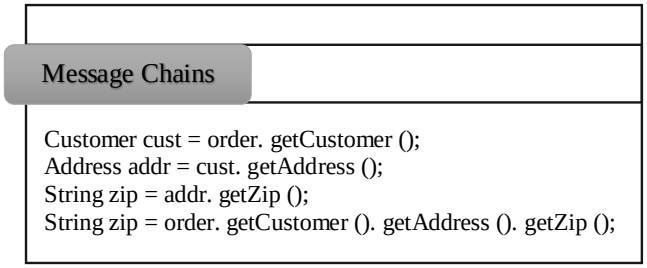


Fig. 1.22. Example of Message Chain code smell.

Management, Smoke and Mirrors, Project Mismanagement, Throw It over the Wall, Fire Drill, The Feud, and E-mail Is Dangerous.

The detailed description of these Anti-Patterns is given in workbook [33]. Some studies also proposed approaches to detect code anti-patterns instead of code smells, since they describe more generic flaws, in this work we use three of them:

- *Blob:* It is also known as God Class. Low cohesion and large size are the characteristics of a Blob class. It is a complex and large class that only uses other classes as data holders and integrates the behavior of a portion of a system.

- *Functional Decomposition:* This code smell is introduced by non-experienced developers. It occurs when a class declares multiple private fields and implements a few of them. i.e. the concept of polymorphism and inheritance associated within the class is scarcely used. This code smell is introduced if a class is designed with the intent to perform a single function.

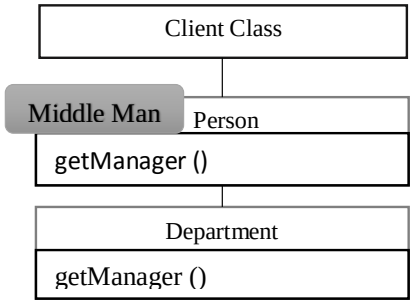


Fig. 1.23. Example of Middle Man code smell.

Feature Envy

```
Public class Customer
{
PrivatePhNumber mbPhone;
        ……
 public String
  getMobilePhNumber()
   {
     return“(”+ mbPhone.getAreaCode()+“)” + mbPhone.    getPrefix( ) + “-” + mbPhone.getNumber( );
   }
}
```

Fig. 1.24. Example of Feature Envy code smell.

- *Spaghetti Code:* A class that uses global variables and declares long methods without parameters, thus, violating the principle of inheritance and polymorphism.

1.3.4 Disadvantages of Code Smell

- Impact on software comprehensibility, modification, and enhancement: It is obvious that the person who develops the software is not the one who maintains it. Furthermore, presence of code smells not only decrease the comprehensibility but also hampers the design and hinder the enhancement and modifications. Thus, the system becomes more complex and even small modifications are difficult to make [41, 46, 30].

- Impact on software quality: It is widely believed that code smells are one of the major threats to the quality of the software. Some researchers claim that code smells have a positive impact on software quality [12] whereas, some

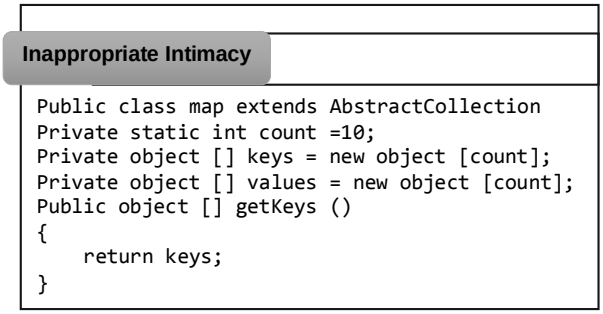


Fig. 1.25. Example of Inappropriate Intimacy code smell.

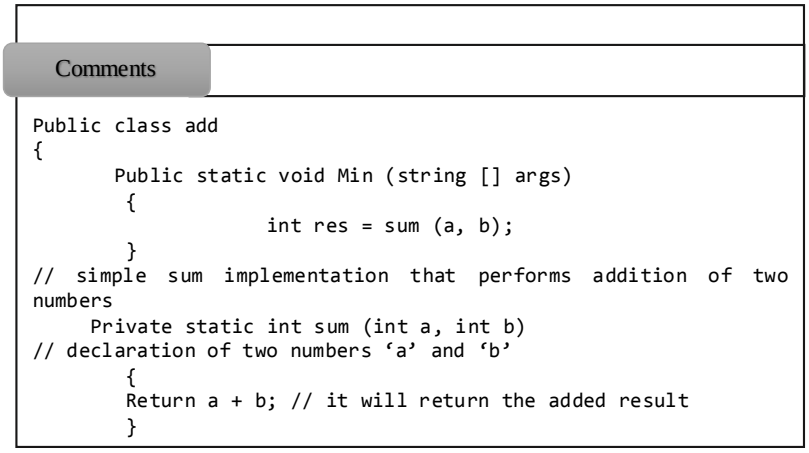


Fig. 1.26. Example of comments code smell.

other researchers found the negative impact of code smells on quality [47]. In addition, some authors also reported the neutral impact of code smells on software quality [48]. The actual impact of code smells on software quality is unclear

- Impact on design: code smells have a bad impact on design which urges the researchers to detect and refactor code smell so as to improve the overall design.

- Bug propagation: If some class/method of source code contains a bug and that class/method is used somewhere else in the programme then the bug will prorogate in that class also. Therefore, code smells raise the possibility of bug propagation [49].

- Higher maintenance cost: Various studies [50, 51, 52] claimed that the

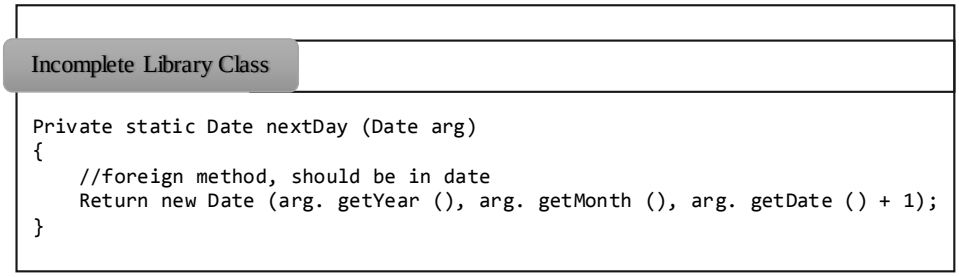


Fig. 1.27. Example of Incomplete Library Class code smell.

presence of code smells increases the post implementation effort [53].

1.4 Detection and Prioritisation of Code Smells

This subsection explains the code smell selection process, need of prioritization, and application of machine-learning in code smell detection.

1.4.1 Code Smell Detection Process

Several techniques employing different methods have been proposed to detect various type of code smells [41, 54]. These code smell detection techniques use the source code or its compiled form in different representations as an input along with code smell specifications. The object oriented software metrics or some other techniques like machine leaning [55], etc are used to match these bad smell specifications with examined source code. The value of these metrics is either calculated by using some third party tool or by directly performing static analysis of the source code. Furthermore, the calculated value of software metrics is used by the smell detector along with bad smell specifications, and instances of code smells are identified as an output. A detailed description of code smell detection process is shown in Fig. 1.28. It is not possible to detect all code smells using static analysis because of dynamic dispatch, run time binding, etc. Therefore, some authors used dynamic analysis based approaches to detect bad smells like Feature Envy [49].

Apart from this, some other techniques also used hybrid of machine-learning and

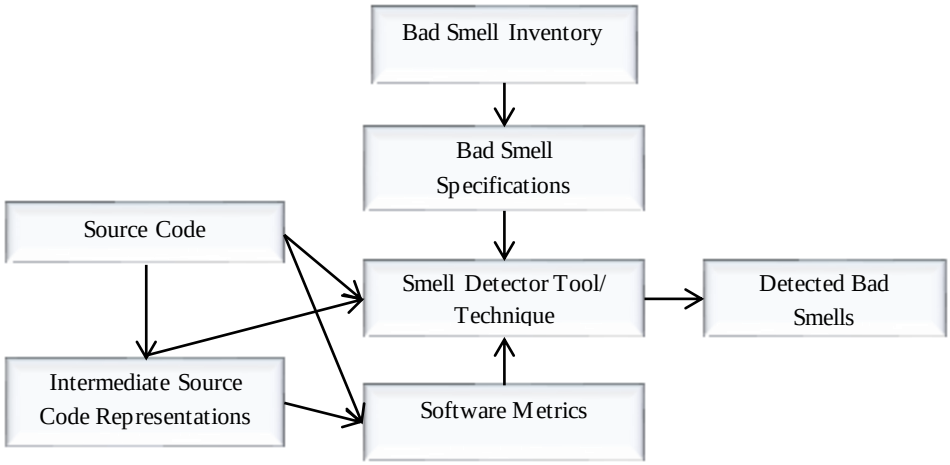


Fig. 1.28. Code smell detection process.

metaheuristic techniques and conjuction of different metaheuristic algorithms for detecting bad code smells [56]. Furthermore, some code smells like Shotgun Surgery, Divergent Change, Parallel Inheritance, etc. cannot be appropriately detected from the program source code. Hence, versioning history information is used to locate these aforementioned bad smells [57]. The information about versioning history can be gathered from Github, Sourceforge etc.

1.4.2 Need of Prioritization of Code Smell

The existence of code smells in the code put a negative impact on software quality. Therefore, several code smell detection techniques [56, 58, 59] have been proposed for the identification of the code smells from the source code. These techniques return the identified code smells which in turn requires a lot of effort for their removal by using appropriate refactoring techniques. But the cost and effort available for removing these code smells is limited. Also, all the code smells are not equally important or relevant to the goals of software or its health. Hence, these code smells should be ranked according to their importance or relevance. Code smell prioritization techniques decide the order in which these smells must be removed, so that good quality software can be obtained in lesser time with less cost and least effort. This will in turn enhance the maintainability of the software.

Identification of code smells is the first activity of the refactoring process. Various automated tools have been proposed to assist the developers in identifying the code smells. But the results yielded by such tools overwhelm the developers. The results may also include many false positive indications too. Thus, it becomes a tedious task for the developer to examine each and every code smell of the system. Additionally, the refactoring effort needed to resolve all the code smells generally exceeds the limited budget that is provided to the developers for carrying out the whole refactoring process.

Moreover, not all code smells are equally severe as compared to others. Fowler [29] has introduced 22 code smells, but it is challenging for developers to decide which code smell should be refactored first. This decision is made according to the harmful impact of code smells on software quality attributes. So, this calls for a need of prioritizing the code smells. Researchers proposed various techniques for code smell prioritization based on the impact of the code smells on the overall quality of the software. Since prioritization is adopted to resolve the deadline and budget issues, some researchers introduced approaches that consider the removal of such code smells that involves low refactoring cost and are easier to refactor.

1.5 Application of Machine-learning Techniques in Code Smell Detection

Many existing code smell detection techniques are rule based [60, 61, 32]. Regardless of good performance shown by these techniques, there are number of challenges that are faced by the researchers in this area. Firstly, these techniques use the combination of Object-Oriented metrics to detect code smell. Different approaches use different metric combinations and each combination has a different threshold value. Even if the same metrics are employed, their threshold value may vary. Thus, with the change in threshold value, the number of detected code smell may increase or decrease. Therefore, the performance of detector is

strongly influenced by the selection of threshold value [62].

Another challenge is related to accuracy of results. Too many false positive or false negatives can be detected as the information related to size, design, domain, and context of analysed dataset is not considered [63]. This increasing volume and variety of information has made it tedious for the programmers to manually detect every possible code smell.

Therefore, in such situations, the need of machine-learning arises. Machine-learning approaches are relatively cheap and fast [64, 65]. They detect and correct errors and bugs without human intervention. Also, these techniques learn from its experience to avoid the defects in future.

1.6 Decision Tree Algorithm

Decision tree algorithm is one of the most practical and widely used method of supervised machine-learning algorithms. It is used to solve both classification as well as regression problems. For prediction and classification supervised training is used which means the classes of each instance is already known in the training set. This approach provides a tree structure where data splits continuously on the basis of certain parameters or conditions. The tree can be generated using three entities: root node, decision node contain test attributes where data splits, and leaves contain class label that gives final outcome or decision. Root node is the highest level node of a tree. After training process, the model is able to classify an instance of unknown class on the basis of path followed from root node to leaf node.

1.6.1 J48 Algorithm

From the family of decision tree algorithms, J48(C4.5) is selected for identification of code smells due to its popularity and accuracy in results. It has been developed by Ross Quinlan [66, 67]. It is an extension of ID3 (Iterative Dichotomiser 3). There is a open source Java implementation of C4.5 in WEKA. The additional

features of J48 includes: a) derivation of rules, b) trees pruning, c) accounting missing values and d) ranges the value of attributes continuously etc. It is one of the best classification algorithm to analyse the data continuously and categorically. Its main objective is to split the data into homogenous class as much as possible in terms of variables to be predicted. J48 allows classification either based on generated rules or on the basis of decision tree. Its objective is to reduce the impurity or uncertainty in data. For handling continuous attributes, it creates a threshold value and split the attributes into two lists, one having values greater than threshold and other containing values less than or equal to threshold. However, for missing values it marked as "?". These missing values are not used in entropy and gain calculation.

The input to algorithm is a set of training (classified) data. Based on this input, it generates a decision tree as an output where each leaf node represents a decision and non-leaf node represents a test. After verifying all the test path from root node to leaf node the decision is represented by a leaf node whether a variable belongs to class or not. Each path from root to leaf is a rule [68, 69]. Once the tree is generated, it is applied to each tuple of database which results into classification of data of that tuple. While generating a tree, C4.5 ignores the missing values. The values of that item can be predicted from the known value of that attribute in other records.

Thus, the classification model generates a tree in top-down fashion. It uses normalized information gain splitting criteria. The information gain of each attribute is judged according to entropy measure. The attribute with highest normalized information gain is selected to make decision. Following this, the best attribute is selected as a root of next recursively constructed subtree. It then, recurses on the partitioned subsets. The basic steps followed in J48 algorithm are as follows:

The advantages of J48 algorithm are described as follows:

Algorithm 1 J48 algorithm

Input: Dataset D
Output: Return N
Create a root node N
If T $\in$ C **then**
    Leaf node=N
    Mark N as class C
    Return N
For i=1 to n
    Calculate information gain
Ta=testing attribute
N.Ta=attribute having highest information gain
If N.Ta == Continuous **then**
    Find Threshold
For each T in splitting of T
    If T is empty **then**
        Child of N is a leaf node
    Else
        Child of N=dtree T
Calculate classification error rate of node N
Return N

- To mitigate the overfitting, J48 inherently employs Single Pass Pruning Process.

- It works with both continuous and discrete data.

- It handles the incomplete data issues effectively.

1.6.2 Performance Metrics

To calculate the gain this algorithm uses Entropy, Information Gain, and Gain Ratio metrics.

- **The Entropy:**

  Entropy measures the disorder in data. The Entropy of Class $\vec{y}$ is calculated as:

$$Entropy(\vec{y}) = -\sum_{j=1}^{n} \frac{|y_i|}{|\vec{y}|} \log\left(\frac{|y_i|}{|\vec{y}|}\right) \tag{1.1}$$

The Entropy of any attribute j of a class y is calculated as:

$$Entropy(j|\vec{y}) = -\frac{|y_j|}{|\vec{y}|}\log\left(\frac{|y_j|}{|\vec{y}|}\right) \tag{1.2}$$

- **Information Gain:**

  The gain of an attribute is calculated by subtracting the Entropy of a class from the Entropy of an attribute.

$$Gain(\vec{y}, j) = Entropy(\vec{y} - Entropy(j|\vec{y})) \tag{1.3}$$

  The information gain metric is useful only in case of small and medium values. While dealing with large values it provides insufficient information. Thus, as a result it becomes difficult to achieve overall gain value.

- **Gain Ratio:**

  To eradicate the shortcoming of dealing with large values in Information Gain, C4.5 utilises Gain ratio to maximise the gain by dividing the gain with overall entropy due to split argument $\vec{y}$ by value of j. Gain Ratio is calculated as:

$$GainRatio(\vec{y}, j) = -\frac{Gain(\vec{y}, j)}{SplitInfo(\vec{y}, j)} \tag{1.4}$$

  where SplitInfo is defined as:

$$SplitInfo(\vec{y}, j) = -\sum_{j=1}^{n} p^{'}\left(\frac{j}{p}\right)\log\left(p^{'}\left(\frac{j}{p}\right)\right) \tag{1.5}$$

  where $p'(j/p)$ is the proportion of elements present at the position p, taking the value of jth test.

- **Precision:**

  Precision is calculated as the ratio of correctly identified bad smells to the total identified bad smells. From the obtained value of the Precision, one

can conclude the likelihood that the identified code is accurate.

$$Precision = \frac{\{(\text{Relevant code smells}) \cap (\text{Detected code smells})\}}{(\text{Detected code smells})} \tag{1.6}$$

- **Recall:**

  Recall represents the fraction of appropriately detected bad smells among the set of manually identified bad smells to find out the count of bad smells that have not been unexploited. From the value of the Recall, one can deduce the probability that an anticipated code smell is identified.

$$Recall = \frac{\{(\text{Relevant code smells}) \cap (\text{Detected code smells})\}}{(\text{Relevant code smells})} \tag{1.7}$$

- $F_{measure}$**:**

  $F_{measure}$ is defined as the harmonic mean of Precision and Recall.

$$F_{measure} = 2 \times |Precision \times Recall / Precision + Recall|\% \tag{1.8}$$

1.7 C5.0 Algorithm

C5.0 is a new generation of J48 or C4.5 machine-learning algorithm with an additional features such as, efficiency, accuracy, less memory, smaller decision tree, and fast speed [70]. Decision trees are generated on the basis of set of training examples and list of possible attributes. C5.0 follows the rules of C4.5 algorithm, i.e., the method of generating a decision tree in C5.0 is similar to that of C4.5 [71]. C5.0 algorithm operates by splitting the data sample on the basis of field having greatest information gain. The sample subset obtained from previous split is split afterward and this process continues till this subset cannot be further split. Lastly, it examines the lowest level split and the subsets with least contribution are rejected. Algorithm 2 shows the steps of C5.0 algorithm followed to construct a

decision tree.

Algorithm 2 C5.0 algorithm

Input: Dataset P, training data and their corresponding class labels, attribute election approach, and attribute list
Output: Decision tree

```
Make a node M
If all samples in P ∈ class D then
    Return M as a leaf node having label as D
If attribute list is unfilled then
    Return M as a leaf node having label as majority class in P
Execute attribute selection approach to obtain the best test attribute
Assign label test attribute to node M
while every output k of test attribute do
        Generate a branch from node M for the condition test attribute = Pk
end while
/*** suppose Pk as the set of data samples in P fulfilling output k ***/
If Pk is vacant then
    Join a leaf node having label as majority class in P to node M
Else
    Join the node provided by this recursive algorithm to leaf node
Return M
```

The advantages of C5.0 over C4.5 algorithm are as follows:

- During classification it can anticipate the most significant and non significant attributes. i.e., it automatically allows removing unhelpful attributes.
- It solves the problem of error pruning and over fitting.
- It also gives acknowledgement on missing data.
- It is faster than C4.5.
- In comparison to C4.5, memory usage is more efficient in C5.0.
- It generates smaller decision tree.
- It produces more accurate results and has lower error rate on unseen cases.

1.8 Problem Statement

A developer should develop software with high quality because software with poor quality requires a lot of time, cost and effort for its maintenance. After software development, the software is passed to maintenance team for maintenance. The maintenance team is different from the team of developers. They may or may not have much or exactly the same knowledge as the development team has, about the source code. So they modify it according to them, which lead to occurrence of code smells in the software. Code smells are not the errors in the program but these can slower down the functionality of the program. Moreover, these smells can reduce the quality and increase the maintenance cost and effort. Consequently, it is very necessary to identify the code smells and to prioritize them according to their impact on software quality.

To detect code smells, researchers proposed many techniques such as metric based, manual approach, machine-learning based, metaheuristic based, etc. However, these approaches suffer from various limitations like error proneness due to human intervention, deciding a threshold value for a metric, etc. Therefore, there is a need to study the efficiency of existing techniques and develop a better approach for code smell detection.

Moreover, existing code smell detection techniques generate a list containing too many code smells. These code smells are not equally harmful for the software in terms of software quality. Without knowing the harmful impact of code smells, developers get reluctant towards removing them by refactoring due to limited time and budget pressure. Hence, a need of code smell prioritization approach arises which can sort the detected smells based on their impact on software quality attributes. This smell prioritization will motivate the developers towards eradicating the harmful smells which in turn will increase the adoption of refactoring practice in software industry.

1.9 Research Objectives

1. To study the existing code smell detection techniques.
2. To design and develop a technique for detection of bad smells using machine-learning.
3. To test and validate the performance of proposed technique.
4. To study the effect of prioritization of code smells on software quality.

1.10 Research Methodology

- Firstly, we studied the existing machine-learning, metaheuristic, and hybrid techniques used to design and propose a bad smell detector that can aid software developers in producing high quality software systems.
- Then, a novel search based hybrid algorithm is proposed and is used in conjuction with J48 and C5.0 to detect various code smells as an output. The sample applications written in Object-Oriented programming language are considered for evaluation of proposed approach. The source code of the sample application along with the set of quality metrics and code smell specifications are supplied as an input to code smell detector.
- After detecting code smells, the optimal ranking of these code smells is produced by using a proposed algorithm.
- In addition, the impact of code smell prioritization is analysed by performing an empirical study.
- Finally, the comparative analysis of the proposed approach is performed with existing approaches in the area of code smell detection and prioritization.

The major steps involved in the code smell detection and prioritization are outlined in the Fig. 1.29

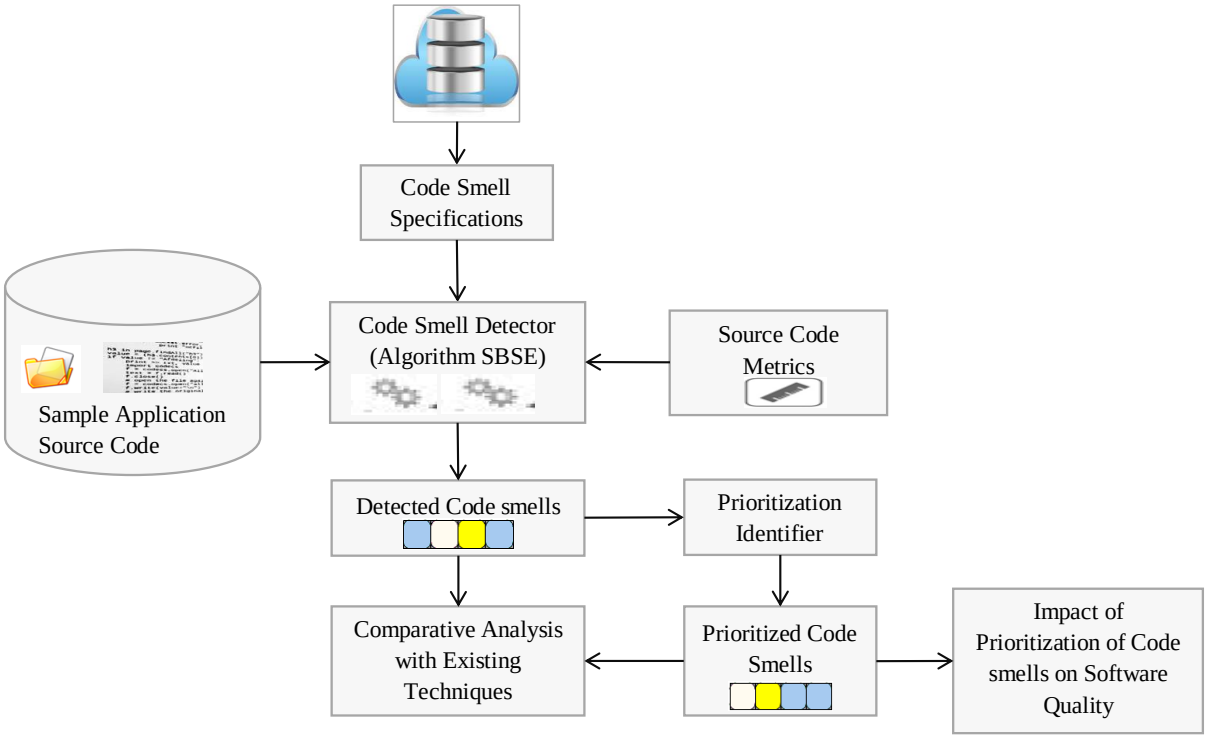


Fig. 1.29. Research methodology.

1.11 Contribution to the Thesis

To fulfill the objectives, this thesis makes three major contributions (see Fig. 1.30). These contributions are described as follows:

Contribution 1: Hybrid Code Smell Detection Approaches

A novel metaheuristic algorithm called Sandpiper Optimisation Algorithm (SPOA) is proposed. SPOA is inspired from the biological behavior of sea birds (i.e., sandpipers). Later, we hybridise it with J48 machine-learning algorithm in order to solve the problem of parameter tuning. The proposed hybrid approach is named as SP-J48. The main purpose of hybridization is to tune the parameters of machine-learning algorithms for efficient selection of their values. In order to detect the optimal code smells, SP-J48 takes open source projects, Object-Oriented metrics, and detection rules based on combination of metrics and their threshold values as an input.

The proposed technique is tested on real-life datasets and its results are compared

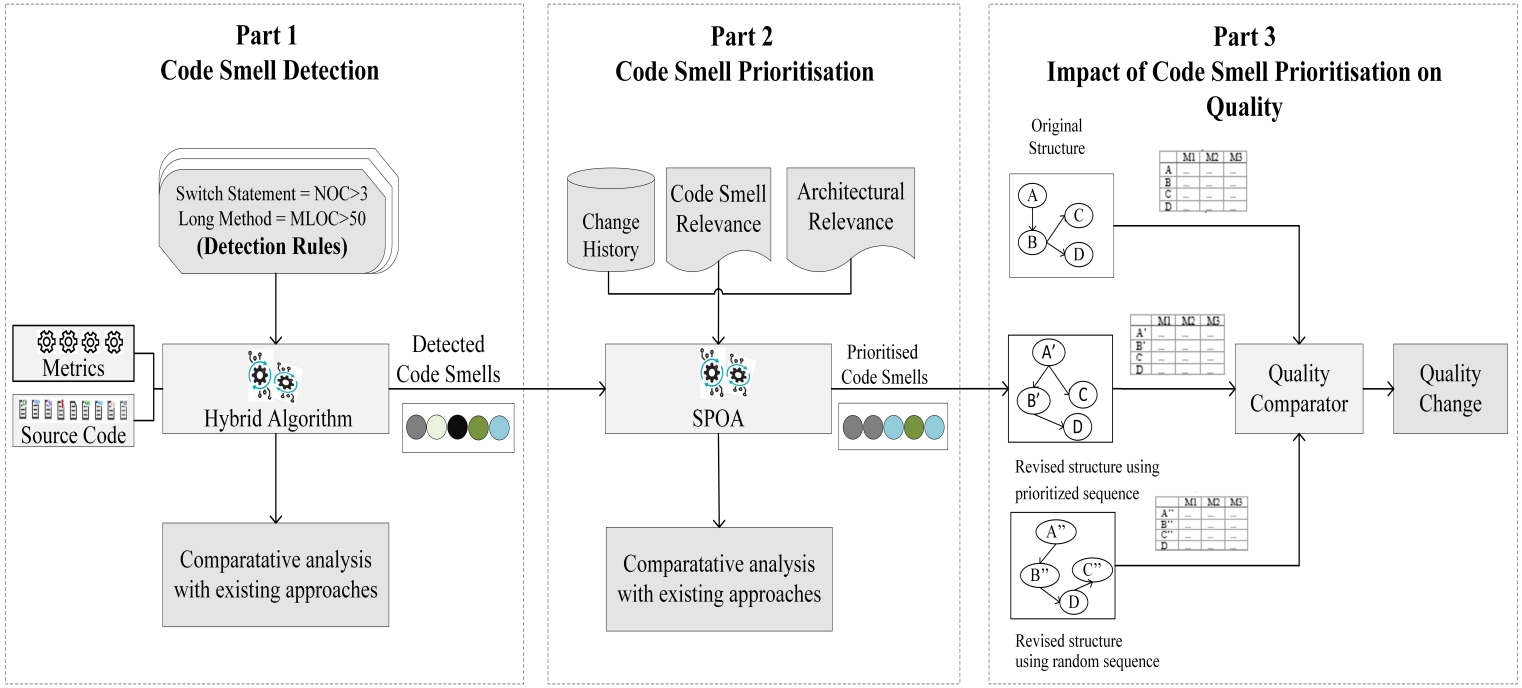


Fig. 1.30. Contribution to the research methodology.

with various state-of-the-art approaches. The outcomes show that the proposed technique performs better than the other approaches with 88% Precision and 79% Recall. Moreover, due to the advantages of C5.0 over J48, it is used in conjunction with SPOA for code smell detection and named it as C5.0-SPOA. The results of C5.0-SPOA is compared with existing approaches and SP-J48. The performance of proposed hybrid approach is evaluated on five Java open source softwares using Precision, Recall, and $F_{measure}$. The obtained results signify that the proposed technique provides better results as compared to existing state-of-the-art techniques and SP-J48 with 91% Precision, 78% Recall, and 82% $F_{measure}$.

Contribution 2: Prioritization of Detected Code Smell

The proposed approach is extended to prioritize the detected set of code smells. Three parameters namely, versioning history, code smell relevance, and architectural relevant classes are used for prioritizing the most severe code smells. Firstly, the frequently refactored classes from previous versions of a software are calculated. Then, from the current version of a software, the classes which are architecturally relevant to the design of a system are considered. Furthermore, the common set of classes among obtained set of classes from above two steps are shortlisted. In addition, manual code smell severity is assigned to the code smells of these classes.

Later, the values obtained from these three parameters are given as an input to the proposed approach in order to generate the rank according to their impact on software quality. The performance of proposed approach is evaluated on Java open source projects by using three performance metrics namely, estimated effort, code smell correction ratio, and severity of fixed code smells.

It is analysed from results that in previous versions of each application, an average of 38% of the total classes has been frequently refactored. Moreover, among those frequently refactored classes, almost half of the classes are architecturally relevant. Furthermore, in the current version for each sample application, an average of 19% of the total classes were architecturally relevant. Therefore, nearly 85% of

refactoring effort can be saved by prioritizing the classes based on the proposed approach and an average of 65% code smells gets corrected with the proposed class prioritization approach. In addition, the severity of every version is reduced with an average of 20 severity score. Overall it has been analysed from the reported results that the proposed approach is able to reduce developers effort by finding the severe code smells which in turn will save developer's time and maintenance cost.

Contribution 3: Analysing the Impact of Code Smell Prioritization on Software Quality Attributes

In this contribution, a case study is performed to empirically analyse the impact of code smell prioritization on software quality attributes. For this, three software versions including the original version, the version generated by removing a random set of code smells, and the version produced by removing the code smells in prioritized order returned by our proposed prioritisation approach are taken. The metric values are calculated for these three versions and compared with each other to quantify the effect of code smells on prioritization. C&K metric suit is taken into consideration to quantify the effect of code smell sequence on cohesion, coupling, complexity, size, and inheritance attributes. The performance of proposed prioritisation approach is evaluated on three open source softwares written in Java.

The results of this analysis show that for Xerces-J, code smells removed in the order generated by our proposed approach reduces the values of LOC, RFC, CBO metrics by 0.35%, 0.24%, and 0.65%, respectively. Whereas, with a random sequence, a reduction of only 0.10%, 0.17% and 0.35% has been achieved. It is also analysed that at 5% level of significance, the code smells refactored in prioritized order are statistically significant than the values obtained after removing code smells in random order.

1.12 Organisation of the Thesis

The thesis is organized as per the following chapters:

Chapter 1: Introduction

This chapter introduces the basic concept of code smell, evolution of code smell, code smell taxonomy, need of code smell detection, code smell prioritization, basics of metaheuristics, and machine-learning. It briefly describes the basic process of code smells detection. It also highlights the problems caused by code smells in a software. This chapter also discusses the challenges faced in detecting code smells while using existing techniques. Furthermore, it also describes the need of prioritization of code smells for refactoring. In a nutshell, this chapter explains the basic concepts of code smell detection, need of machine-learning in code smell detection, code smell prioritization along with problem definition and research objectives.

Chapter 2: Literature Review: State-of-the-art

This chapter provides a detailed review of the existing literature. Firstly, code smell detection techniques using numerous methods such as manual approach, metric based approach, machine-learning algorithms, optimization techniques, techniques using combination of search based methods as a hybrid, etc. are discussed. Later, the state-of-the-art in the area of code smell prioritization techniques is covered. In last, the research gaps identified from the existing literature are outlined. Further, the research problem is designed on the basis of these research gaps.

Chapter 3: SP-J48: A Hybrid Approach for Code Smell Detection

In this chapter, firstly a novel metaheuristic optimization algorithm is introduced. This algorithm is named as Sandpiper Optimization Algorithm (SPOA) which mimics the behaviors of sandpipers in nature. SPOA is tested on real-life benchmark test problems to show its efficiency and effectiveness. Further, SPOA

algorithm is hybridized with J48 machine-learning algorithm, named as SP-J48. It is used to find the code smell from open source software efficiently. The performance is evaluated on Precision, Recall, and $F_{measure}$ metrics.

Chapter 4: Code Smell Detection using C5.0-SPOA

In this chapter, C5.0 machine-learning algorithm is used in conjunction with SPOA, named as C5.0-SPOA. The proposed technique is used for code smell detection and prioritization. The performance of C5.0-SPOA is evaluated in terms of Precision, Recall, and $F_{measure}$ on Java based Open source software projects.

Chapter 5: Impact of Code Smell Prioritization on Software Quality

This chapter discusses the empirical study performed to analyse the effect of code smell prioritization on internal software quality. Cohesion, coupling, inheritance, size, and complexity metrics are used to analyse the impact of prioritization of code smells. Statistical testing is also performed to validate the obtained results.

Chapter 6: Conclusions and Future Scope This chapter concludes the thesis by providing a brief overview of proposed approaches and providing an insight into the future work. The thesis is concluded on the basis of contributions made concerning proposed approaches for code smell detection, code smell prioritization and analysing the impact of code smells prioritization on quality. The summarization of future directions in the area of software engineering, machine-learning, and metaheuristics, will enlighten the research community regarding research areas in the field of artificial intelligence.

Chapter2

Literature Review: State-of-the-art

"I still believe in the possibility of a model of reality, that is to say, of a theory, which represents things themselves and not merely the probability of their occurrence." By Einstein

This chapter describes the state-of-art in the area of code smells. The literature is reviewed from two fields: code smell detection and code smell prioritization. In 1999, several researchers started developing code smell detection techniques which either perform static analysis or dynamic analysis to detect code smells which are discussed in detail. In addition to this, literature in the field of code smell prioritization is also discussed in following subsections.

2.1 Static Analysis based Code Smell Detection Techniques

In this section, existing research on various static code smell detection approaches like Manual, Metrics, Symptom, Probabilistic, Visualization, Cooperative, Machine-learning, Search based and Hybrid approaches for detection of code smells are discussed.

- **Manual Based Approach**

  Earlier manual code smell detection techniques were used to detect code smells. Travassos et al. [72] provided manual inspection and reading

guidelines to detect design defects. These techniques require human involvement and are time consuming. Although human involvement reduces uncertainties in detection process, but these techniques are infeasible for detecting code smells from large software systems.

- **Metric Based Approach**

  Several metric based approaches have been used to detect bad smells. Ganea et al. [60] introduced a continuous quality assessment tool named InCode which warns a developer about the occurrence of a code smell as it appears with the contextualized explanation of that smell. InCode detects Code Duplication, Feature Envy, God Class, and Data Class smells using metric based rules. This tool helps the developers in addressing the design problems earlier as the contextualized information guides the developer in correcting the detected smells. The proposed tool is evaluated on five open source systems (OSS) namely JHotDraw 7.6, Eclipse JDT 3.8.1, AgroUML 0.34, Maven 3.1.1 and Vuze 5301.

  Marinescu [32] defined metric based detection strategies for identifying design flaws at class, method and subsystem level. Detection strategies are obtained by combining various metrics using set operators. Metric values are compared against relative and absolute thresholds to detect around ten design flaws. Munro [61] provided metric based heuristics for detecting Temporary Field and Lazy Class bad smells in open source Java systems. The author justified the choice of metrics and thresholds for detecting code smells by performing an empirical study on Graph Tool and hotel booking system.

  Similar to the aforementioned approaches, Salehie et al. [73] proposed a classification process based framework to detect design flaws which uses two levels. At first level, hot spots are detected using primitive classifiers. At second level, these hot spots are classified using metric based heuristics to detect Refused Bequest, Shotgun Surgery, Feature Envy and God Class code

smells. To evaluate their proposed approach, they have used Jave Based Open Source Software (JBOSS).

Fard and Mesbah [74] provided JSNOSE tool for detection of thirteen JavaScript code smells namely Switch Statement, Long Message Chain, Excessive Global Variables, Refused Bequest, Nested Callback, Closure Smell, Long Method/Function, Large Object, Coupling JS/HTML/CSS, Empty Catch, Lazy Object, LongParameterList and Unused/Dead code. The projected tool was evaluated on eleven web applications.

Yuanfang et al.[75] proposed DRSpace, a new model of software architecture that can be used in source code to evaluate the architecture implemented They proposed the concept of ArchRoots, as the first exploration of using DRSpace is to reveal the architectural impact on software quality. ArchRoots attract and propagate errors to multiple files, and their detection algorithm. They used 15 projects with varying sizes, domains, and ages, and observed that 1) a bug-prone design rule can also make DRSpace error-prone for a large number of files inside it; 2) the effects of architecture connections between bug-prone files are severe and persistent over time. Their research also shows that the effects of architectural links between bug-prone files are persistent.

Palomba et al.[76] examined whether code smells identified using textual information are as hard to detect and refactor as structural smells or if they follow a dissimilar pattern during software development. They conducted two distinct studies examining how developers respond on instances of code smell of the same kind, but identified either by TACO or by (but not both) structural-based methods. They used 20 open source softwares to identify five code smells namely, Blob, misplaced code elements: i.e., Feature Envy and Misplaced Class, Long Method, and Promiscuous Package.

They concluded that industrial developers typically consider textually detected code smells as real design problems and as hazardous as structural

code smells, even if they do not exceed the thresholds of any structural metrics. Moreover, textually detected code smells are more prone to be resolved through refactoring operations or enhancement activities.

- **Symptom Based Approach**

  Symptom based techniques use different symptoms or notations for identification of code smells. Moha et al.[77] projected an approach called DÉCOR to detect various code and design smells. Domain analysis of the code was performed and design smells were specified using domain specific language (rule cards). These algorithms are used to identify four design smells (namely Swiss Army Knife, Spaghetti Code, Blob and Functional Decomposition) and fifteen code smells automatically. The approach is validated by independent engineers using eleven open source systems on the basis of Precision and Recall for assessing whether the detection algorithms correctly detected the code and design smells.

  Marinescu and Ratiu [78] proposed PRODEOOS tool for the identification of bad smells in C++ and Java. They used coupling, cohesion, data abstraction and complexity metrics to detect four code smells namely God class, Data Class, Refused Bequest and Shotgun Surgery. The tool was evaluated on two case studies of industrial projects.

- **Probabilistic Based Approach**

  Probabilistic approaches making use of fuzzy logic rules is an another way to detect bad smells. Rao and Reddy [79]proposed an approach which uses design propagation probability matrices (DCPP matrix) to identify two types of code smells namely Divergent Change and Shotgun Surgery. The authors evaluated their approach on three small examples. Nitin [46] implemented Java Smell Detector (JSD) tool for the identification of Java code smells. Static analysis was performed to identify code smells such as Switch Case, Middle Man, Long Method, Data Class and LongParameterList. The tool is evaluated on 28 graduate projects.

- **Visualization Based Approach**

  Visualization based approaches use semi-automatic process for detecting code smells. Simon et al.[80] proposed Crocodile tool to identify four code smells namely Large class, Feature Envy, Lazy Class and Inappropriate Interface using manual analysis. The tool was evaluated on VRML software system. Murphy et al. [81] provided a Strench Blossom tool to detect four code smells namely Long Method, Data Clump, Large Class and Feature Envy from Java projects. The evaluation of tool was done on Java Libraries and Vuze.

- **Cooperative Based Approach**

  Cooperative based code smell detection techniques detect various code smells with greater accuracy by performing different activities in a cooperative way. Abdelmoez et al. [82] proposed a technique to detect four code smells such as Long Method, LongParameterList, Message Chain and Empty Catch, and the severity of these bad smells was estimated using threshold values of metrics. They used two parallel algorithms to increase the speed of the search and to decrease the space for search. Genetic programming was used as basis for the applied algorithms.

  The technique was evaluated on three case studies of diet care C# program namely Appointments.cs, Person.cs and Jobs.cs. These Co-operative based techniques are comparatively new, and used to increase performance and accuracy for detection of code smell.

- **Machine-learning Based Approach**

  Machine-learning techniques [83] provide another alternative to detect code smells without human intervention and errors. Due to difficulty in finding threshold values of metrics for detection of code smells, lack of consistency between different identification techniques, and developers subjectiveness in identifying code smells from these tools, numerous techniques using machine-learning algorithms have been projected by researchers in literature

[84]. The first technique in this field was proposed in 2005. In 2005, Kreimer [85] developed an Eclipse Plug-in named IYC for identification of design flaws in software systems. Classical approaches like abstract interpretation or data and control flow were used to compute values of Object-Oriented metrics (like cohesion, complexity, size and coupling). C4.5, a decision tree based machine-learning technique was used to detect five types of code smells namely Feature Envy, Long Method, Lazy Class, Big Class and Delegator. Among these five code smells, only Long Method and Big Class were evaluated on two case studies viz., WEKA and IYC systems.

Similarly, Vaucher et al.[86] used Naive Bayes approach to detect Blob code smell from two open source softwares namely Xerces v2.7.0. and Eclipse JDT. Khomh and Vaucher [87] extended the DÉCOR tool proposed by Moha et al. [77] by transforming their rules into Bayesian belief network (BBN) for the detection of Blob anti-patterns. This approach was evaluated on two Java open source systems namely GanttProject 1.10.2 and Xerces 2.7.0 and the results of BBN were compared with DÉCOR tool.

In 2011, Khomh et al. [55] extended their previous approach [87] and proposed a new Bayesian approach called BDTEX based on GQM (Goal Question Metric) to detect anti-patterns. The authors used developers feedback and the definitions of anti-patterns rather than rules based intermediate nodes representation to detect three anti-patterns called Spaghetti Code, Functional Decomposition, and Blob. The proposed approach was evaluated on GanttProject 1.10.2 and Xerces 2.7.0 open source softwares and the results of BDTEX were compared with DÉCOR.

Later, Oliveto et al. [88] proposed an ABS (Antipattern identification using B-Splines) technique using B-Splines to detect same smells from two medium sized Java open source systems. The accuracy of their proposed approach was compared with DÉCOR [77] and BBN [87]. Furthermore, Hassaine et al. [89] used immune-inspired approach called IDS (Immunebased Detection

Strategy) for detection of same code smells from GanttProject v1.10.2 and Xerces v2.7.0 datasets. The proposed approach was compared with DÉCOR [77] and BBN [87].

Maneerat and Muenchaisri [90] used 7 different machine-learning algorithms to detect Lazy Class, Feature Envy, Middle Man, Message Chains, Long Method, LongParameterLists and Switch Statement code smells. The authors used 7 industrial datasets from previous literature. In their work, they compared the results of Random forest with 6 other machine-learning algorithms. The results reported that due to less code smell prediction rate, some algorithms such as J48, VFI, and Naive Bayes, are not appropriate for code smell prediction as the prediction rate of these algorithms is less than 90%.

Danphitsanuphan and Suwantada [91] developed an Eclipse Plug-in called 'Bad Smell Detection Tool (BSDT) to detect the location of code smells in software systems. The authors used class and method level Object-Oriented software metrics to detect seven bad smells namely Large Class, Lazy Class, Data Class, Parallel Inheritance Hierarchy, Long Method, LongParameterList and Switch Statement. After detecting bad smells, Naive Bayes data mining technique was used to analyse the relationships between structural bugs and bad smells. The system was evaluated on two Java software systems viz. projects Movie rental program and electricity calculating program.

Maiga and Ali [59] proposed SVMDetect approach to detect anti-patterns in Java source code using support vector machines as machine-learning technique. SVMDetect is able to identify four anti-patterns namely Swiss Army Knife, Spaghetti Code, Blob, Functional Decomposition. The authors concluded that proposed approach outperforms DETEX [77] by evaluating it on three case studies of Java open source softwares namely Xerces 2.7.0, AgroUML 0.19.8 and Azureus 2.3.0.6. Later, Maiga et al. [92] extended their SVMDetect [59] approach by taking into account expert's feedback and

named it as SMURF. A Support Vector Machine algorithm was used to detect four kinds of anti-patterns like Spaghetti Code, Swiss Army Knife, Blob and Functional Decomposition. This approach was manually validated on 3 case studies of Java open source softwares namely Xerces 2.7.0, AgroUML 0.19.8 and AZureus 2.3.0.6 and the results were compared with DETEX [77] and BDTEX [55].

The authors showed that projected approach can be applied on intra as well as inter systems, and correctness of SMURF is better as compared to BDTEX and DETEX. Fu and Shen [93] used association rule mining techniques to detect three types of code smells (Duplicated Code, Shotgun Surgery, and Divergent Change) from five open source softwares namely, Eclipse, jUnit, Guava, Closure-Compiler and maven. The results of technique are compared with SonarQube [94], Décor, jDeodorant and DCPP [79].

Palomba et al. [95] used association mining rules and projected a technique called HIST (Historical Information for Smell deTection) for identification of five code smells namely Blob, Divergent Change, Shotgun Surgery, Feature Envy, and Parallel Inheritance by extracting change history information from versioning systems. The proposed technique was evaluated on eight Java open source projects such as Apache Ant, jEdit, five projects of Android APIs and Apache Tomcat. Moreover, the performance of HIST was compared with JDeodorant and DÉCOR and concluded that apart from detecting code smells it also highlights the frequently changing code smells.

Later, Palomba et al. [96] re-evaluated their approach [95] on another 20 open source systems. The proposed approach was evaluated on two empirical studies: firstly, HIST identifies the code smell that are not identified by the comparative techniques, and to access the accuracy it was evaluated on 20 Java projects. Secondly, they investigated the extent to which code smells can be detected by HIST reflects designer's understanding of poor design and implementation choices. They involved 12 open source software

project developers and reported that more than 75% of detected code smells are actually a design or implementation problems.

Fontana et al. provided [97] an approach based on machine-learning algorithms for detection of code smells. They authors used different metrics for detection of different smells. Furthermore, if metrics used were same, their threshold values were different. An experiment was performed on 76 systems of Qualitas Corpus and targeted to provide vital dataset for code smell detectors to be used as a basis for future standards.
Six different machine-learning algorithms (J48, Random Forest, Naive Bayes, JRip, SMO, and LibSVM) and large set of metrics (such as class, method, package, project level plus standard metrics as size, complexity, cohesion and coupling) were used to detect four code smells namely, God Class, Data Class, Long Method, and Feature Envy. At class level, God Class and Data Class were detected whereas at method level Long method and Feature Envy were detected.
The performance of J48, Random Forest, JRip and SMO were better as compared to Naive Bayes on feature envy and data class code smells. In addition, Fontana et al. [62] used 32 different machine-learning algorithms to detect four types of code smells such as Large Class, Data Class, Feature Envy and Long method in order to identify a best algorithm.

Recently, Di Nucci et al. [64] highlighted some issue of Fontana et al. [62] study and replicate his work by using machine-learning techniques for code smell detection that points the issue of metric distribution in smelly and non-smelly instances of considered softwares. This issue was exploited in the study provided by Fontana et al. They have replicated the study on same datasets using same machine-learning algorithms. In their findings, they concluded that the issue was due to the usage of specific dataset instead of machine-learning algorithms actual capabilities.

Therefore, the problem of code smell detection using machine-learning algorithms is still unsolved. Thus, it is still an open research area and more research is needed in this field. Some researchers have adopted machine-learning techniques for the detection of particular type of code smell, i.e., code clone (a.k.a. Duplicate Code).

Wang et al. [98] used Bayesian Networks to detect Duplicate Code from two large C# projects of Microsoft. The authors considered two scenarios: conservative scenario and aggressive scenario, and reported that their approach is useful for both the scenarios. This approach approves 52% to 60% harmless clones and block 48% to 67% harmful clones.

Later, White et al. [99] employed recurrent neural network (RtNN) technique to detect same code smell from 8 open source Java softwares namely, ANTLR 4, Apache Ant 1.9.6, ArgoUML 0.3.4, CAROL 2.0.5, dnsjava 2.0.0, Hibernate 2, JDK 1.4.2, and JHotDraw 6. They have considered all four types of clone and reported that their technique detected file and method level pairs mapping of all types of clone and, is also feasible in representation and fragments for clone detection.

$$DatasetSize = \begin{cases} Small, & KLOC(thousandLOC) < 50 OR Classes < 200 \\ Medium, & 50 \leq KLOC \leq 250 OR 200 \leq Classes \leq 1000 \\ Large, & KLOC > 250 OR Classes > 1000 \end{cases} \tag{2.1}$$

Furthermore, the results of proposed approach are compared with Deckard [100] and concluded that the proposed learning based approach is more suitable for clone detection comparative to state-of-art practices.

Table 2.1 shows code smell detection techniques using machine-learning algorithms. The guidelines provided by Radjenovic et al. [101] are followed to classify the studies into small, medium, and large sub-dimensions based

on their dataset sizes in terms of number of classes (or files) and number of Lines of Code (LOC) as shown in Equation (2.1).

- **Metaheuristic Based Approach**

  Many researchers [102] have contributed their research in the field of detecting code smells using metaheuristic techniques. They considered the detection of code smells as a search problem. Marcus and Maletic [103] proposed an approach to identify code clones of linked list ADT in C source code. The proposed approach used latent semantic indexing as information retrieval technique and performs static analysis on software systems to determine semantic resemblances.

  This structural and semantic information of source code was used by clustering based PROCSSI system [104] to detect high level concepts in clones. This approach uses minimal spanning tree clustering algorithm to detect two code smells namely Duplicate Code and Inappropriate Interfaces. The projected system was evaluated on 2.7 version of Mosaic system.

  Mihancea and Marinescu [105] used genetic algorithm to detect God Class and Data Class code smells from seven Java and C++ based student projects. To detect the code smells using metric based approaches, the researchers established proper threshold values of metrics using genetic algorithms.

  Kessentini et al. [106] used Genetic Programming to propose an automatic technique to detect the rules for Blob, Spaghetti Code and Functional Decomposition code smells. Detection rules are defined as an integration of threshold and metric values. The researchers evaluated their proposed approach on OSS namely GanttProject v1.10.2 and Xerces-J v2.7.0 on the basis of Precision and Recall. For rule extraction, the proposed approach was further compared with Particle Swarm Optimization (PSO) [107], Harmony Search (HS)[108], Simulated Annealing (SA)[109], and DÉCOR. The authors concluded that their approach outperforms DÉCOR and provide better results with an average of 75% to uncover known code smells.

In 2013, Mansoor et al. [110] used Multi-Objective Genetic Programming (MOGP) approach to find best metric combination to generate code smell detection rules. The proposed technique was evaluated on the releases of five OSS (Apache Ant, ArgoUML, Gantt, Azureus, and Xerces-J) to detect three types of code smells such as Blob, Spaghetti Code, and Functional Decomposition. Furthermore, the proposed approach is compared with Nondominated Neighbor Immune Algorithm (NNIA) and multiobjective optimization with artificial immune systems (MOAIS) [111] using Hypervolume and Inverted Generational Distance metrics.

In addition, the proposed approach is compared with Random search (RS), mono-objective GP and two more existing state-of-art techniques [112, 113] on the basis of Precision and Recall. The results reported that MOGP and MOAIS are better than random forest. Also, the proposed technique outperforms the state-of-art mono-objective techniques with an average of 91% of Recall and 86% of Precision.

Later, Mansoor et al. [114] extended their previous approach [110] to detect five kind of code smells namely Blob, FE, SC and DC and FD using same dataset. The researchers evaluated their approach on seven Java open source softwares and concluded the their Multi-Objective Genetic Programming technique outperforms existing techniques.

Boussaa et al. [115] used Competitive Co-Evolutionary Algorithm (CCEA) to detect three kind of code smells namely Blob, Spaghetti Code and Functional Decomposition. They involved two populations simultaneously, one is to generate the detection rules using combination of metrics in order to increase the inclusion of code smell examples. The another population helps in maximizing the number of generated code smells which are excluded by the detection rules. The proposed approach was evaluated on different versions of four OSS namely ArgoUML v0.26 , Xerces v2.7, Ant-Apache v1.5 and Azureus v2.3.0.6.

To evaluate the performance, the results of proposed technique were compared with Genetic Programming [112] and Artificial Immune System [113] on the basis of Precision and Recall. The reported findings show that the proposed technique provides better results as compared to two single population based metaheuristics approaches.

Similarly, Ouni et al. [116] used genetic programming to contemplate a technique that automatically generate the rules for defect detection. The technique was evaluated on six Java OSS. Furthermore, the performance of proposed technique was compared with DÉCOR and SA, and reported that the proposed technique provides promising results as compared to DÉCOR.

Kessentini et al. [82] used parallel evolutionary algorithm (P-EA)to generate the code smell detection rules. They considered nine OSS to detect eight types of code smells and compared their proposed technique with Genetic Algorithm (GA), Genetic Programming (GP) and Random Search (RS). Furthermore, the proposed technique is also compared with DÉCOR and JDeodorant [117]. The authors concluded that the proposed approach performs better than existing approaches.

Sahin et al. [118] used search based technique and proposed BLOPs (Bi-Level Optimization Problems) tool for detection of code smells. The proposed approach involves two level of optimization. At upper level, they generated bad smell detection rules using Genetic Programming approach and at lower level, they used Genetic algorithm for code smell detection. Quality metrics (WMC, NAS, NOC, RFC, AIF, LCOM, PF, CC, DIT, NA, NC, AH, CBO, NLC and MH) were employed to detect four code smells namely Blob, Feature Envy, Spaghetti Code and Data Class and, three design smells such as Functional Decomposition, LongParameterList and Lazy Class. The proposed tool was evaluated on nine OSS and one industrial datasets.

The performance of proposed BLOP technique is compared with existing state-of-art techniques including Genetic Programming (GP) [112],

co-evolutionary GA [115] and concluded that BLOP outperforms the competitive approaches. Moreover, they compared the proposed approach with DÉCOR [77] and concluded that it outperform DÉCOR in case of Precision.

In 2016, Ghannem et al. [119] used Genetic Algorithm to detect three types of code smells from four Java open source softwares. The proposed approach is compared with the previous work of Ghannem et al. [120] and DÉCOR. Code smell detection techniques using metaheuristic approaches are tabulated in Table 2.6.

- **Hybrid Approach**

  Due to parameter tuning issue in machine-learning techniques, several authors employed metaheuristic techniques in conjunction with machine-learning algorithms to detect code smells.

  Amorin et al. [121] confirmed the findings of Kreimer [85] by using C5.0 decision tree algorithm to detect 12 code smells (Antisingleton, God Class, ClassDataShouldBePrivate, Complex Class, Large Class, Lazy Class, Long Method, LongParameterList, Message Chain, Refused Parent Bequest, Speculative Generality and Swiss Army Knife) on four Java OSS likely Eclipse, Mylyn, ArgoUML, and Rhino.

  Genetic Algorithm is used in conjunction with C5.0 to automatize the metrics selection. The results of their technique were then compared with rule based tools, Bayesian Beliefs Networks, and Support Vector Machine (SVM) technique. The observed comparison results indicate that the performance of decision tree when used together with genetic algorithm is even much better than comparative algorithms.

  Furthermore, Fontana et al. [62] performed the survey of different machine-learning algorithms used for identification of bad smells in order to identify a best algorithm. They selected 32 distinct machine-learning algorithms such as decision tree (J48 , JRIP , RANDOM FOREST), NAIVE

BAYES, support vector machines (SMO , LIBSVM), and, ADABOOST etc. and experimented them on four types of bad smells namely Data Class, Large Class, Feature Envy and Long method. An effective and straightforward Grid Search [122] was used to search the best parameter setting.

Grid search was chosen for two main reasons: firstly, the approach which do not perform exhaustive parameter search employing heuristics or approximations are not considered as safe because of their inability to prove the superiority of experimented combinations over the unexplored ones. Secondly, due to independent setting of each parameter, it can be easily parallelized. They used 74 Object-Oriented systems of Qualitas Corpus and validated 1986 samples manually. The authors concluded that the best performances were obtained by Random Forest and J48 machine-learning algorithm whereas poorest performances were achieved by SVM. This survey can help the researchers in selecting the best technique for code smell detection.

Saranya et al. [123] used a combination of Genetic Algorithm and Particle Swarm Optimization techniques and proposed a hybrid approach based on Euclidean distance called EGAPSO for detection of model level code smells. The proposed approach detects code smells on the basis of set of defect examples and similarity between considered softwares instead of relying on threshold values of metrics. The projected approach detects five code smells from three open source projects.

Furthermore, to analyse the effectiveness of EGAPSO, it is compared with existing state-of-art approaches such as GA [119], DÉCOR [77], P-EA [82] and MOGP [110]. The reported findings prove the effectiveness of EGAPSO in comparison to existing approaches.

Similarly, in 2018, Kacem and Bouassida [124] used a combination of both supervised and unsupervised machine-learning techniques and proposed a hybrid approach for code smell detection employing Auto-encoder algorithm

[125] in conjunction with Artificial Neural Network (ANN) algorithm.

They used mean squared error (MSE) to tune the parameters of auto-encoder learning method and Grid search to optimize the parameters of ANN classifier.

Furthermore, the proposed technique was evaluated on 74 Qualitas Corpus softwares. The efficiency of contemplated approach was evaluated on the basis of Precision and Recall and, concluded that hybrid approach outperforms the basic classifiers. Table 2.8 enlists the code smell detection techniques using hybrid approaches.

2.2 Dynamic Analysis Based Techniques

In these techniques, the behavior of the code is analysed after executing it based on test cases or user scenarios. It is more accurate than static analysis as it examines the actual behavior of source code rather than simply examining the source code. So, due to these advantages researchers proposed efficient code smell detection techniques based on dynamic analysis.

Kumar and Chhabra [49] proposed a two level dynamic analysis based approach to detect Feature Envy code smell. At the first level, dynamic analysis is performed to locate methods that can be Feature Envy. These suspect methods are further analysed at second level using dynamic analysis to assure whether they are feature Envy or not. The approach is evaluated on two open source software packages named SWT example and Textarea package from jEdit software. The results of proposed approach is compared with inCode [58] and JDeodorant [126]. The obtained findings shows that the results gathered through proposed technique are more accurate and less time consuming.

Liu et al.[127] proposed an approach for adapting thresholds dynamically and automatically. In this work authors firstly, analyse the necessity of automatic threshold adaption for code smell detection. Secondly, they suggested an

approach that would automatically and dynamically adjust thresholds. Finally, they validated their proposed approach on five open source applications such as, AutoFlight, DirBuster, Java3D Modeler, PDF Split and Merge, DavMail to identify four code smell namely, long methods, clones, feature envy, and inappropriate intimacy. The evaluation results suggested that their proposed approach is effective and it outperforms the traditional tuning machine.

2.3 Review on Code Smell Prioritization Techniques

Once the code smells are detected, it is necessary to prioritize them for refactoring purpose. Numerous researchers have contributed in the area of designing code smell prioritization techniques in order to provide refactoring opportunities according to their impact on software quality attributes.

Sae et al. [128] used context relevance index (CRI) to prioritize code smells. Firstly the authors used inFusion tool to detect 24 code smells. Furthermore, the authors used CRI to rank the detected code smells from four OSS namely, ArgoUML, Jabref, jEdit, and muCommander. The accuracy of proposed technique is then compared with severity based prioritization approach on the basis of Precision. The results shows that context based code smell prioritization approach gives more relevant outcomes as compared to severity based techniques.

Steidl and Eder [129] prioritize Code Clones and Long Method smells on the basis of refactoring recommendations. They firstly detected these two maintainability defects using ConQAT tool [130] and then identified refactoring opportunities for removing these defects. They proposed various heuristics to prioritize these code smells and evaluated them on seven industrial and six open source softwares.

Vidal et al. [131] proposed SpIRIT tool for identification and prioritization of bad smells on the basis of three parameters such as relevance of code smell, history of component modification and modifiability scenarios. SpIRIT automatically identifies ten code smells (God Class, Brain Class, Tradition Breaker, Data Class,

Brain Method, Shotgun Surgery, Feature Envy, Refused Bequest, Disperse Coupling and Intensive Coupling) using various Object-Oriented software metrics.
These code smells are then prioritized on the basis of aforementioned criteria to help the developers in refactoring the most prioritized smells with least effort. The proposed approach was evaluated on Subscribers DB Application and Beef-Cattle Farm Simulator Java applications. Later, the authors extended their tool [131] and renamed it as JSpIRIT [132] which combines different strategies for detecting agglomerations of smells and provided the prioritization strategies for ranking these smells.

SpIRIT tool was used to detect the code smells from Mobile Media, Health Watcher Java projects. Later, the authors [133] extended this approach and prioritized the detected code smells on the basis of change history and architectural relevance. The approach is evaluated on four projects namely Mobile Media, Health Watcher, SubscriberDB and Apache OODT. The results of each software was compared with the other to conclude that their approach helps the developers to analyse the critical code smells.
Based on association with faults, Zhang et al. [134] prioritized six code smells namely, Duplicate Code, Data Clumps, Middle Man, Message Chains, Switch Statements and Speculative generality. The Copy/Paste Detector (CPD) function of PMD tool [135] was used to identify Duplicate Code and other five bad smells were identified using Pattern based code bad smells detection tool. The Fault history was captured for Eclipse and Apache software systems from Concurrent Versions System (CVS) and Subversion (SVN) repositories, respectively using JIRA tool. After finding fault history, their association with code smells were used to prioritize the bad smells.

Yang et al. [136] use code clone detection tool to detect code clones manually form source code written in C language. The authors developed Fica tool using machine-learning algorithms for individual users to rank the code clones detected by CDT based on historical decision. The results of CDT i.e., 105 clone sets of

bash4.2 were used to evaluate the system.

Fontana et al. [137] used smell intensity index to prioritize code smells. The JCodeOdor tool [138] was proposed to detect six code smells namely God Class, Shotgun Surgery, Message Chain, Data Class, Dispersed Coupling and Brain Method using metric based rules. Smell intensity index was integrated with the tool which helps in prioritizing the most critical smells. The intensity index distribution for these six smells was evaluated on 74 systems of Qualitas Corpus [139].

In addition, Fontana ans Zanoni [140] used machine-learning algorithms to classify the severity of code smells. They detected four code smells likely God Class, Long Method, Data Class, and Feature Envy from 76 systems of Qualitas Corpus and ranked the code smells according to their severity. Table 2.10 shows the studies ranking the code smells according to their harmful impact on software quality.

Table 2.1: Machine-learning based code smell detection techniques.

Author and Year	Tool/ Technique	Code smells	Algorithm	Dataset	Dataset language	Dataset type	Dataset size			Metrics used
							S	M	L	
Kreimer, 2005	IYC	Long Method, God Class, Feature Envy, Delegator, and Lazy Class	C4.5 decision trees	IYC and WEKA	JAVA	OSS and Industrial	1	1	0	No. of instance variables of a class, Median of the number of statements of all methods of a class, Median of complexities of all methods of a class, Number of internal connected components of a class, Number of external connected components of a class, NOP, NOLV
Vaucher et al., 2009	N.I.	Blob	Naive Bayes technique	Xerces v2.7.0. and Eclipse JDT	XML, JAVA	OSS	N.I.	N.I.	N.I.	NMD, NAD, LCOM5, NAM, ControllerClassrule
Khomh et al., 2009	N.I.	Blob	Bayesian Belief Networks	GanttProject v1.10.2 and Xerces v2.7.0	JAVA	OSS	1	1	0	NMD, NAD, LCOM5, NAM, ControllerClassrule
Oliveto et al., 2010	ABS	Blob	B-Splines	N.I.	JAVA	OSS	0	1	0	
Hassaine et al., 2010	IDS	Blob, Functional Decomposition, and Spaghetti Code	Immune-inspired approach	GanttProject v1.10.2 and Xerces v2.7.0	JAVA	OSS	1	1	0	NAD, NADextended, NCP, NMA, NMD, NMDextended, NOM, PP
Khomh et al., 2011	BDTEX	Blob, Functional Decomposition, and Spaghetti Code	Bayesian Belief Networks	GanttProject v1.10.2 and Xerces v2.7.0	JAVA	OSS	1	1	0	NMD, NAD, LCOM5, NAM, ControllerClassrule
Maneerat and Muenchaisri, 2011	N.I.	Lazy Class, Feature Envy, Middle Man, Message Chains, Long Method, Long Parameter Lists, Switch Statement	Random Forest, Naive Bayes, Logistic, IBI, IBk, VFI, and J48	7 datasets from the previous literatures	N.I.	Industrial	N.I.	N.I.	N.I.	Size:NA, NC, NM, NO, ACT, COMP, NS Complexity RFC, WAC, WMA, NP Coupling CBC Encapsulation: AHF, AIF, CF, MHF, MIF, PF Inheritance: DIT, NOC, NAI, NOI Class :C_PARAM Employment Diagrams: D_APPEAR Relationship: ABSTR_R, ASSOC_R, DEPEND_R

Table 2.2: Machine-learning based code smell detection techniques. (*Continued*).

Author and Year	Tool/ Technique	Code smells	Algorithm	Dataset	Dataset language	Dataset type	Dataset size			Metrics used
							S	M	L	
Danphitsanuphan and Suwantada, 2012	BSDT	Large Class, Long Method, Parallel Inheritance Hierarchy, Long Parameter List, Lazy Class, Switch Statement, and Data Class	Naive Bayes and Association Rules	Movie rental program and electricity calculating program	JAVA	Industrial	2	0	0	NOM , LOC, DIT , PAR, MLOC, WMC, NOF, NCS, VG, LCOM
Maiga et al., 2012	SVMDetect	Functional Decomposition, Spaghetti Code and Swiss Army Knife	SVM approach	ArgoUML v0.19.8, Azureus v2.3.0.6, and, Xerces v2.7.1	JAVA	OSS	0	1	2	LOC, NAD, NADextended, NMA, NMD, NMDextended, NOM
Maiga and Ali (2012)	SMURF	Functional Decomposition, Spaghetti Code and Swiss Army Knife	SVM approach	ArgoUML v0.19.8, Azureus v2.3.0.6, and, Xerces v2.7.0	JAVA	OSS	0	1	2	LOC, NAD, NADextended, NMA, NMD, NMDextended, NOM
Wang et al, 2012	N.I.	Duplicate Code	Bayesian Networks	Two large industrial software projects from Microsoft	C#	Microsoft project (Industrial)	0	0	2	Size: LOC Coupling: Number of Invocations, Number of Library Invocations, Number of Local Invocations, Number of Other Invocations, Number of Field Accesses Process: History Features (Existence-Time, Number of Changes, Number of Recent Changes, File Existence-Time, Number of File Changes, Number of Recent File Changes), Destination Features (Whether it is a Local Clone, Fine Name Similarity, Masked File Name Similarity, Method Name Similarity, Sum of Parameter Similarities)

Table 2.3: Machine-learning based code smell detection techniques. (*Continued*).

Author and Year	Tool/ Technique	Code smells	Algorithm	Dataset	Dataset language	Dataset type	Dataset size			Metrics used
							S	M	L	
Palomba et al., 2013	HIST	Blob, Divergent Change, Shotgun Surgery, Feature Envy and Parallel Inheritance	Association Rule Mining (Apriori)	Apache Ant, jEdit, five projects of Android APIs and Apache Tomcat	JAVA	OSS	0	5	3	Change history of systems
Fontana et al., 2013	N.I.	God Class, Data Class, Feature Envy, Long Method	Support Vector Machines (SMO, LibSVM), Decision Trees (J48), Random Forest, Naive Bayes, Jrip	76 open source softwares of Qualitas Corpus	JAVA	OSS	N.I.	N.I.	N.I.	LOC, LOCNAMM, NOM, NOPK, NOCS, NOMNAMM, NOA, CYCLO, WMC, WMCNAMM, AMWNAMM, AMW, MAXNESTING, WOC, CLNAMM, NOP, NOAV, ATLD, NOLV
Palomba et al., 2014	HIST	Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy	Association Rule Mining (Apriori)	20 open source softwares	JAVA	OSS	5	9	6	Change history of systems

Table 2.4: Machine-learning based code smell detection techniques. (*Continued*).

Author and Year	Tool/ Technique	Code smells	Algorithm	Dataset	Dataset language	Dataset type	Dataset size			Metrics used
							S	M	L	
Arcelli Fontana et al., 2015		Data Class, Large Class, Feature Envy and Long Method.	32 different ML algorithms like:J48 , JRIP , RANDOM FOREST , NAIVE BAYES, SMO , LIBSVM and ADABOOST etc	74 open source softwares of Qualitas Corpus	JAVA	OSS	19	38	17	Size: LOC, LOCNAMM, NOM, NOPK, NOCS, NOMNAMM, NOA Complexity: CYCLO, WMC, WMCNAMM, AMWNAMM, AMW, MAXNESTING, WOC, CLNAMM, NOP, NOAV, ATLD, NOLV Coupling: FANOUT, FANIN, ATFD, FDP, RPC, CBO, CFNAMM, CINT, CDISP, MaMCL, MeMCL, NMCS, CC, CM Encapsulation: LAA, NOAM, NOPA Inheritance DIT, NOI, NOC, NMO, NIM, NOII Cohesion: LCOM5, TCC Others: NODA, NOPVA, NOPRA, NOFA, NOFSA, NOFNSA, NONFNSA, NOSA, NONFSA, NOABM, NOCM, NONCM, NOFM, NOFNSM, NOFSM, NONFNABM, NONFNSM, NONFSM, NODM, NOPM, NOPRM, NOPLM, NONAM, NOSM
Amorin et al., 2015	N.I.	Antisingleton, GodClass, Class data Should Be Private, Complex Class, Large Class, Lazy Class, Long Method, Long Parameter List, Message Chain, Refused Parent Bequest, Speculative Generality , and Swiss Army Knife	C5.0 decision trees	Eclipse , Mylyn, ArgoUML and Rhino	JAVA	OSS	0	1	3	LOC, LOC_1, MLOCsum, NAD, NADExtended, NMA, NMD, NMDExtended, NOM, WMC, McCabe, NOF, NOP, NOParam, Vgsum, WMC1, WMC_New, CBO, RFC, CA, CE, CAM, ACAIC, ACMIC, DCAEC, DCC, DCMEC, IR, NCM, NOTI, RFC_New, Connectivity, DAM, DIT, NOC, MFA, AID, CLD, DIT_1, ICHClass, NMI, NMO, NOA, NOC_1, NOD, NOH, NOPM, LCOM, LCOM3, LCOM1, LCOM2, LCOM5, CohesionAttributes, NPM, MOA, IC, CBM, AMC, DSC, NOTC

Table 2.5: Machine-learning based code smell detection techniques. (*Continued*).

Author and Year	Tool/ Technique	Code smells	Algorithm	Dataset	Dataset language	Dataset type	Dataset size			Metrics used
							S	M	L	
Fu and Shen, 2015	N.I.	Duplicated code, shotgun surgery, and divergent change	Association Rule Mining (Apriori or FP-growth)	Eclipse, jUnit, Guava, Closure-Compiler, and maven	JAVA	OSS	N.I.	N.I.	N.I.	WMC, NOM, CBO
White et al., 2016	N.I.	Duplicate Code	The recurrent neural network (RtNN)	ANTLR 4, Apache Ant 1.9.6, ArgoUML 0.34, CAROL 2.0.5, dnsjava 2.0.0, Hibernate 2, JDK 1.4.2, JHotDraw 6	JAVA	OSS	7	1	0	Tokenized source code, e.g., terms of the class and programming constructs used in the class
Kaur et al., 2017	SVMCSD	LC, LM, DC and FE	SVM approach	Xerces v 2.7.0 and ArgoUML v0.19.8	JAVA	OSS	n/a	n/a	n/a	n/a
Dario Di Nucci et.al, 2018	N.I.	Data Class and God Class, Feature Envy and Long Method	32 different ML algorithms like:J48 , JRIP , RANDOM FOREST , NAIVE BAYES, SMO , LIBSVM and ADABOOST etc.	74 open source softwares	JAVA	Open Source Software	19	38	17	Size: LOC, LOCNAMM, NOM, NOPK, NOCS, NOMNAMM, NOA Complexity: CYCLO, WMC, WMCNAMM, AMWNAMM, AMW, MAXNESTING, WOC, CLNAMM, NOP, NOAV, ATLD, NOLV Coupling: FANOUT, FANIN, ATFD, FDP, RPC, CBO, CFNAMM, CINT, CDISP, MaMCL, MeMCL, NMCS, CC, CM Encapsulation: LAA, NOAM, NOPA Inheritance DIT, NOI, NOC, NMO, NIM, NOII Cohesion: LCOM5, TCC Others: NODA, NOPVA, NOPRA, NOFA, NOFSA, NOFNSA, NONFNSA, NOSA, NONFSA, NOABM, NOCM, NONCM, NOFM, NOFNSM, NOFSM, NONFNABM, NONFNSM, NONFSM, NODM, NOPM, NOPRM, NOPLM, NONAM, NOSM

Table 2.6: Metaheuristic based code smell detection techniques.

Author and Year	Tool/ Technique	Code smells	Algorithm	Dataset	Dataset language	Dataset type	Dataset size			Metrics used
							S	M	L	
Marcus and Maletic, 2001	n/a	Code clone	Minimum Spanning tree	Mosaic 2.7	C	Industrial	1	0	0	Software System, Cluster, File, A source code document
Mihancea and Marinescu, 2005	n/a	God Class, Data Class	Genetic Algorithm	Experiment based on student projects	JAVA, C++	Student project (Academic)	7	0	0	ATFD, WMC, TCC, NOM, NOPA, WOC
Kessentini et al., 2011	n/a	Blob, SC and FD	Genetic Programming	GanttProject v1.10.2 , Xerces-J v2.7.0	JAVA	OSS	0	2	0	DIT, WMC, CBO, LOCCLASS, LOCMETHOD, NAD, NMD, LCOM5, NACC, NPRIVFIELD
Mansoor et al., 2013	MOGP	Blob, SC, and FD	Multi-Objective Genetic Programming (MOGP)	ArgoUML v0.26, ArgoUML v0.3, Xerces v2.7, Ant-Apache v1.5, Ant-Apache v1.7.0, Gantt v1.10.2, Azureus v2.3.0.6	JAVA	OSS	0	2	5	WMC, RFC, LCOM, NA, AH, MH, NLC, CBO, NAS, NC, DIT, PF, AIF, NOC
Boussaa et al., 2013	CCEA	Blob, SC, and FD	Competitive Co-Evolutionary Algorithm (CCEA)	ArgoUML v0.26 , Xerces v2.7, Ant-Apache v1.5 , Azureus v2.3.0.6	JAVA	OSS	0	1	3	The number of lines of code in a class, number of lines of code in a method, number of attributes in a class, number of methods, lack of cohesion in methods, number of accessors, and number of private fields
Ouni et al., 2013	n/a	Blob, SC, and FD	Genetic Programming	GanttProject v1.10.2, Quick UML v2001, AZUREUS v2.3.0.6, LOG4J v1.2.1, ArgoUML v0.19.8, and Xerces-J v2.7.0.	JAVA	OSS	2	2	2	Metrics proposed by Gaffney 1981 [gaffney1981metrics]

Table 2.7: Metaheuristic based code smell detection techniques. (*Continued*).

Author and Year	Tool/ Technique	Code smells	Algorithm	Dataset	Dataset language	Dataset type	Dataset size			Metrics used
							S	M	L	
Kessentini et al., 2014	P-EA	Blob, FE, SC and DC , FD, LPL, LzyC and SS	Parallel evolutionary algorithm (PEA)	ApacheAnt v1.5.2, ApacheAnt v1.7.0, Xerces-J V2.7.0, GanttProject v1.10.2, Rhino v1.7R1, Log4J v1.2.1	JAVA	OSS	2	5	2	DIT, WMC, CBO, LOCCLASS, LOCMETHOD, NAD, NMD, LCOM5, NACC, NPRIVFIELD
Sahin et al., 2014	n/a	Blob, FE, SC and DC and, three design smells such as FD, LPL and LzyC	Genetic Programming, Genetic algorithm	ApacheAnt v1.5.2, ApacheAnt v1.7.0, Xerces-J V2.7.0, GanttProject v1.10.2, Rhino v1.7R1, Log4J v1.2.1	JAVA	OSS	2	5	2	WMC, RFC, LCOM, CC, NA, AH, MH, NLC, CBO, NAS, NC, DIT, PF, AIF and NOC
Ghannem et al., 2016	n/a	Blob, FD and DC	Genetic Algorithm	GanttProject v0.10.2 , LOG4J v1.2.1, Xerces v2.5 and ArgoUML v0.18.1	JAVA	OSS	0	3	1	NA, NPvA, NPbA, NProtA, NM, NPvM, NPbM, NPrtM, Nass, NAgg Ndep, Ngen, NAggH NGenH, DIT, HAgg
Mansoor et al., 2017	MOGP	Blob, FE, SC and DC and FD	Multi-Objective Genetic Programming (MOGP)	ArgoUML v0.26, ArgoUML v0.3, Xerces v2.7, Ant-Apache v1.5, Ant-Apache v1.7.0	JAVA	OSS	0	2	5	WMC, RFC, LCOM, NA, AH, MH, NLC, CBO, NAS, NC, DIT, PF, AIF, NOC

Table 2.8: Hybrid techniques for code smell detection.

Author and Year	Tool/ Technique	Code smells	Algorithm	Dataset	Dataset language	Dataset type	Dataset size			Metrics used
							S	M	L	
Arcelli Fontana et al., 2015		Data Class, Large Class, Feature Envy, and Long Method	32 different ML algorithms such as J48, JRIP, RANDOM FOREST, NAIVE BAYES, SMO, LIBSVM, ADABOOST and so on	74 open source softwares of qualitas corpus	JAVA	OSS	19	38	17	Size: LOC, LOCNAMM, NOM, NOPK, NOCS, NOMNAMM, NOA Complexity: CYCLO, WMC, WMCNAMM, AMWNAMM, AMW, MAXNESTING, WOC, CLNAMM, NOP, NOAV, ATLD, NOLV Coupling: FANOUT, FANIN, ATFD, FDP, RPC, CBO, CFNAMM, CINT, CDISP, MaMCL, MeMCL, NMCS, CC, CM Encapsulation: LAA, NOAM, NOPA Inheritance DIT, NOI, NOC, NMO, NIM, NOII Cohesion: LCOM5, TCC Others: NODA, NOPVA, NOPRA, NOFA, NOFSA, NOFNSA, NONFNSA, NOSA, NONFSA, NOABM, NOCM, NONCM, NOFM, NOFNSM, NOFSM, NONFNABM, NONFNSM, NONFSM, NODM, NOPM, NOPRM, NOPLM, NONAM, NOSM
Amorin et al., 2015		Antisingleton, GodClass, Complex Class, Large Class, Lazy Class, Long Method, Long Parameter List, Message Chain, Refused Parent Bequest, Speculative Generality, and Swiss Army Knife	Decision trees and Genetic Algorithm	Eclipse, Mylyn, ArgoUML, and Rhino	JAVA	OSS	0	1	3	LOC, LOC_1, MLOCsum, NAD, NADExtended, NMA, NMD, NMDExtended, NOM, WMC, McCabe, NOF, NOP, NOParam, Vgsum, WMC1, WMC_New, CBO, RFC, CA, CE, CAM, ACAIC, ACMIC, DCAEC, DCC, DCMEC, IR, NCM, NOTI, RFC_New, Connectivity, DAM, DIT, NOC, MFA, AID, CLD, DIT_1, ICHClass, NMI, NMO, NOA, NOC_1, NOD, NOH, NOPM, LCOM, LCOM3, LCOM1, LCOM2, LCOM5, CohesionAttributes, NPM, MOA, IC, CBM, AMC, DSC, NOTC

Table 2.9: Hybrid techniques for code smell detection. (*Continued*).

Author and Year	Tool/ Technique	Code smells	Algorithm	Dataset	Dataset language	Dataset type	Dataset size			Metrics used
							S	M	L	
Saranya et al., 2017	EGAPSO	Blob, Functional Decomposition, Spaghetti Code, Data Class, and Feature Envy	Genetic Algorithm and Particle Swarm Optimization	GanttProject, Log4j, and Xerces-J	JAVA	OSS	0	2	0	NA, NPvA, NpbA, NprotA, NM, NPvM, NPbM, NprotM, NAss, Ngen
Kacem and Bouassida, 2018	n/a	God Class, Data Class, Feature Envy, and Long Method	Auto-encoder and Artificial Neural Network Algorithm	74 open source softwares of qualitas corpus	JAVA	OSS	19	38	17	Size: LOC, LOCNAMM, NOM, NOPK, NOCS, NOMNAMM, NOA Complexity: CYCLO, WMC, WMCNAMM, AMWNAMM, AMW, MAXNESTING, WOC, CLNAMM, NOP, NOAV, ATLD, NOLV Coupling: FANOUT, FANIN, ATFD, FDP, RPC, CBO, CFNAMM, CINT, CDISP, MaMCL, MeMCL, NMCS, CC, CM Encapsulation: LAA, NOAM, NOPA Inheritance DIT, NOI, NOC, NMO, NIM, NOII Cohesion: LCOM5, TCC Others: NODA, NOPVA, NOPRA, NOFA, NOFSA, NOFNSA, NONFNSA, NOSA, NONFSA, NOABM, NOCM, NONCM, NOFM, NOFNSM, NOFSM, NONFNABM, NONFNSM, NONFSM, NODM, NOPM, NOPRM, NOPLM, NONAM, NOSM

Table 2.10: Code smell prioritization techniques.

Author and Year	Tool/ Technique	Code smells	Dataset	Dataset language	Dataset type	Dataset size			Performance metrics
						S	M	L	
Zhang et al., 2011	PMD	Duplicate Code, Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man	Eclipse , Apache Commons	JAVA	Open Source Softwares	0	1	1	N/A
Yang et al., 2012	CDT (for detection)/ Fica (for prioritisation)	Code clone	Bash 4.2	C	Industrial	N/A	N/A	N/A	N/A
Steidl et al., 2014	ConQAT	Code Clones and Long Method	ArgoUML, Jabref, jEdit, ConQAT, Subclipse, Tomcat, and 7 industrial systems	JAVA	Industrial and OSS	0	9	4	Precision
Vidal et al., 2015	Java Smart Identification of Refactoring opportunITies (JSpIRIT)	BrC, TB, DC, GC, BrM, SS, FE, RB, DisC, and IC	Application	JAVA	Industrial	N/A	N/A	N/A	N/A
Fontana et al., 2015	JCodeOdor	God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling, and Message Chain	74 systems of Qualitus Corpus	JAVA	Open Source Softwares	N/A	N/A	N/A	N/A

Table 2.11: Code smell prioritization techniques. (*Continued*).

Author and Year	Tool/ Technique	Code smells	Dataset	Dataset language	Dataset type	Dataset size			Performance metrics
						S	M	L	
Vidal et al., 2016	Smart Identification of Refactoring opportunITies (SpIRIT)	BrC, TB, DC, GC, BrM, SS, FE, RB, DisC, and IC	Beef-cattle farm simulator (BCFS), subscribers DB application , SweetHome3D,	JAVA	Industrial and OSS	1	2	N/A	Precision
Vidal et al., 2016	JSpIRIT	Dispersed Coupling and Feature Envy	Mobile Media, Health Watcher, SubscriberDB, and Apache OODT	JAVA	N/A	1	2	1	Precision
Vidal et al., 2017	JSpIRIT	N/A	Mobile Media and Health Watcher	JAVA	N/A	N/A	N/A	N/A	N/A
Fontana and Zanoni, 2017	iPlasma	God Class, Data Class, Long Method, and FE	76 systems of Qualitus Corpus	JAVA	Open Source Softwares	N/A	N/A	N/A	N/A
Sae et al., 2018	inFusion	Blob Class, Data Class, Distorted Hierarchy, God Class, Refused Parent Bequest, Schizophrenic Class, Tradition Breakers, Blob Operation, Data Clumps, External Duplication, Feature Envy, Intensive Coupling, Internal Duplication, Message Chains, Shotgun Surgery, Sibling Duplication	ArgoUML, Jabref, jEdit, muCommander	JAVA	Open Source Softwares	1	1	2	Precision

2.4 Research Gaps and Motivation

After studying the existing literature, the following gaps in research are outlined:

- The Precision of metric based code smell detection techniques directly depend on right selection of source code metrics, their specifications and threshold values. The standard threshold value of these metrics is still unknown and unreliable. Also, a small set of code metrics cannot be used to detect all code smells because of distinct nature of code smells [61, 62].

- Manual and visualization based code smell detection techniques are human centric, time consuming, non-scalable and error prone.

- Symptom based techniques have no agreement for defining standard symptoms with similar specification. Due to different specifications of same symptoms, their Precision is low [41].

- Many existing tools parse the source code directly to detect code smells. But it is not feasible on large system because these techniques work on the basis of statistical data gathering.

- Even if cohesion and coupling are the only major parameters considered in existing literature while evaluating the code smells, no research focus on the type of cohesion and coupling among the classes and methods, which should have been taken care of specially while assigning the threshold values to metrics.

- Majority of the developers used genetic, simulated annealing and hill climbing algorithms for identification of bad smells from source code. Variations of genetic and hill climbing algorithms as NSGA, NSGA II, K-mean Clustering, Decision tree and Random forest are widely used. Whereas, only a few experimented search based software detection techniques either for code smell detection or for their prioritization [82, 105, 106, 97, 64].

- Vastly used tool for code smell detection are jDeodorant, DÉCOR, iPlasma, which accepts the source code for identification of code smell. A few tools are available for code smell detection apart from jDeodorant, iPlasma, etc. which are also not updated and are obsolete. An updated tool is required for code smells detection and, prioritization according to their impact on software quality [77, 117, 41].

- Recent studies have focused on Machine learning techniques for code smell detection as these approaches provide better results in terms of precision and recall, in comparison to other approaches. However, these techniques suffer from the issue of parameter tuning [121].

- To address the issue of parameter tuning, researchers have used machine learning algorithms in conjunction with optimization algorithms. But this approach is adopted by few researchers only. For example, Amorim et. al [121] used genetic algorithm in combination with C5.0 decision tree algorithm, but the exploration and exploitation of genetic algorithm is low as compared to newly proposed optimization techniques. [121, 118].

Therefore, from the retrospective literature survey and aforementioned identified research gaps, it is observed that among all the code smell detection approaches, machine-learning techniques help the researchers in improving code smell detection process by deeply analysing the kind of dependencies between classes, methods, attributes and assigning weights to them by maintaining low coupling and high cohesion. In addition, these approaches outperform other state-of-art approaches in terms of precision and recall but suffers from the appropriate selection of metrics set. This is because the researchers manually need to adjust the set of metrics to get better results. This problem can be eradicated through the use of optimization approaches [121]. Therefore, there is a need of a machine-learning based code smell detection technique that can be used in conjunction with better optimization algorithm in order to automatically select

significate set of metrics which in turn can provides better results.

This desideratum of an efficient approach motivated me to propose a new technique for code smell identification which can assist the industrialists in eradicating the more harmful code smells which will in turn maintain the quality of software as well as reduce the cost and effort in maintaining the software. Moreover, it can also helps the practitioners and researchers to conduct study comparisons.

Chapter3

SP-J48: A Hybrid Approach for Code Smell Detection

"The purpose of computing is insight, not numbers." By Richard W. Hamming

3.1 Overview

The main aim of this chapter is to automatically identify code smells using search based technique and avoid incomprehensive, hardly expanded, and changeable program structure. Code smells are the symptoms of poor design and implementation choices. These can increase change-proneness, fault-proneness, and hinder understandability of software projects. Code smells are interpreted in a different way based on engineer's experience and perception. Due to inaccurate definitions of code smells it is difficult for the developers to decide whether a particular piece of code is a smell or not.

Many techniques have been proposed in literature but some of these do not provide the clear understanding of how these code smells are detected. So in this Chapter, a decision tree algorithm is used to detect code smells from open source softwares. To identify the code smells the values of object oriented metrics are used as these rules can be easily discussed and validated. Therefore, J48 (decision tree) machine-learning algorithm is applied in the software metrics of three Java

Kaur, A., Jain, S. and Goel, S., "**SP-J48: a novel optimization and machine-learning-based approach for solving complex problems: special application in software engineering for detecting code smells**", *Neural Computing and Applications*, Springer, (2019), PP. 1-19. **(SCI, Impact Factor: 4.77)**

based open source softwares namely, GanttProject, Xerces-J, and Log4j to identify five code smells namely, Blob, Feature Envy, Data Class, Functional Decomposition, and Spaghetti Code. The definition of these code smells is given in Chapter 1.

Furthermore, to solve the issue of parameter settings of J48, it is hybridized with our proposed novel bio-inspired metaheuristic technique called SPOA. This hybrid technique is named as SP-J48 algorithm. The results of proposed approach are compared with existing code smell detection techniques. This chapter discusses the code smell detection using J48, proposed hybrid approach, detection of code smell using proposed hybrid approach.

3.2 Code Smell Detection using J48

In order to identify code smells automatically, object-oriented metrics are given as an input to decision tree machine-learning technique. The basic concept of code smell detection is shown in Fig. 3.1. The detection of code smells from source

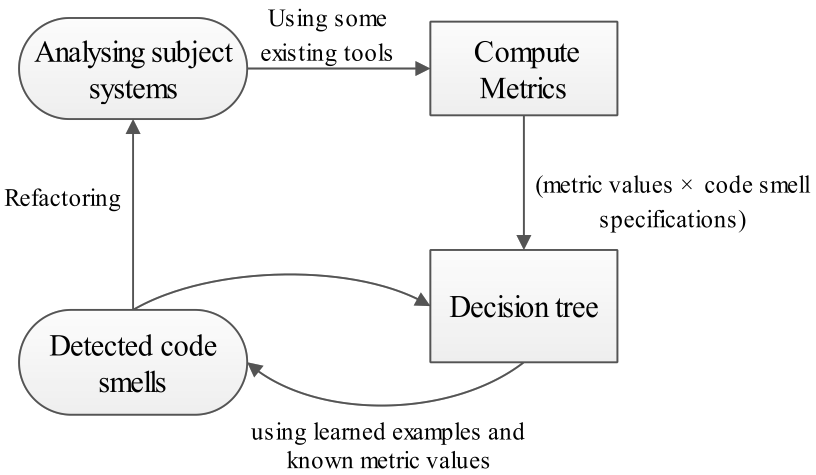


Fig. 3.1. Flow diagram of code smell detection.

code is also a classification problem. It is decidable whether a program location (class or method) contains a smell or not. Each code smell is formed by a set of object-oriented metrics. In decision tree algorithms, these metrics are used as an source attributes and target attributes defines whether a code smell exists or not.

The set of program properties are assigned to each code smell. Object-oriented metrics indicate the properties of these program. Cohesion, Coupling, Complexity, Inheritance, and Size metrics are taken as an input. Thus, a model is formed that provides a detection strategy for each code smell. Metric values are computed by static program analysis. Hence, these calculated metric values are given as an input to machine-learning algorithm. Further, the machine-learning algorithm is trained with the examples of known code smells and metrics values. The proposed model learns from these examples to detect the code smells on the basis of specific metrics values.

The proposed model consists of two parts namely, Training and Testing phase. In training phase, the presence of code smells in the program location is decided manually based on selected program location. The measured values of program location along with the decisions regarding the code smells forms a training set which is used to construct initial decision tree. In testing phase, user defines the location of the program which is to be analysed. Therefore, in this model all unknown locations of a system are measured. Thus, every decision trees of all corresponding code smells is applied for measurement. Moreover, every program location affected by code smell is shown to the user.

Furthermore, the user validates each instance and decides whether to accept or reject the presumed code smell. In both the situations, the program location will be added as an another example to the training set. The training set residue unchanged if the user completely ignores the validation of an instance. Both training and testing phases are entirely independent of each other. The user can add on more suspicious parts of a code to the training set while the testing process is going on. Fig. 3.2 shows the both training and testing phase with adaption of J48 for code smell detection. From the metric values, several rules are generated. These rules are used by J48 to generate a decision tree, similar to one shown in Fig. 3.3. These generated decision trees further assist in detecting the presence of code smells in a particular class. Table 3.1 shows an example of set of instances

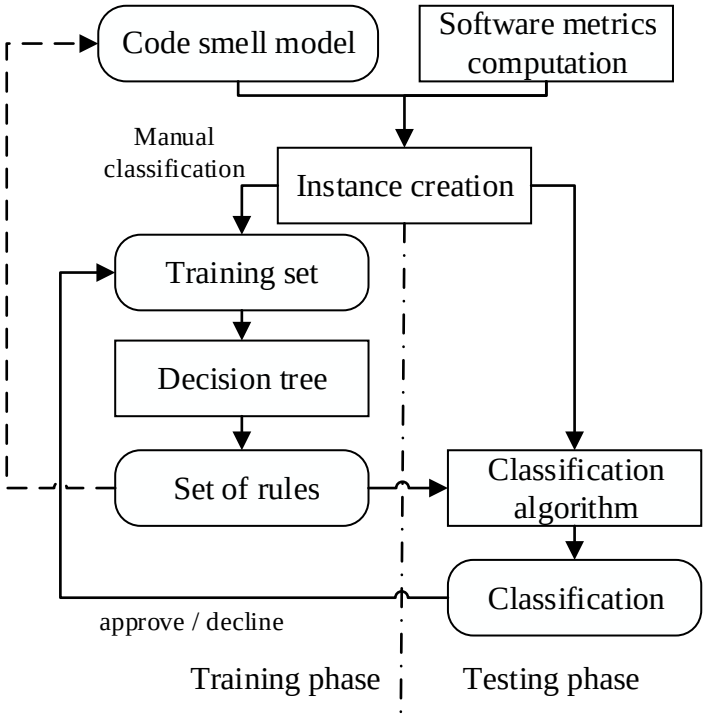


Fig. 3.2. Code smell detection using J48.

for different classes of a program. In this Table, the set of attributes and number of training examples have been shorten. For simulation purpose, only 16 training examples and attributes are tabulated in Table 3.1. The classification model of this type needs atleast 100 training examples to construct a decision tree with sufficient accuracy. The derived decision tree of a code smell as a classification is shown in Fig. 3.3.

It is observed from the tree that non-leaf nodes consist of metrics and leaf nodes represent the results predicted by proposed model to analyse the existence of code smells. There are a total of 62 metrics in a complete metric set but only most significant metrics are selected by J48 on the top of tree. Whereas, less significant metrics appears at lower level of a tree and become more specialised as the tree deepens.

Furthermore, by depicting the decision tree the instances which are not classified earlier, i.e., the instances which does not have the value for target attribute gets

Table 3.1: Set of training examples for Blob code smell.

S. No.	LOC Class	LOC Method	WMC	NAD	NOM	Target attribute Blob
1	750	129	62	5	100	Yes
2	950	150	30	20	30	Yes
3	1325	100	40	5	10	No
4	500	115	70	12	22	Yes
5	1200	175	45	10	50	Yes
6	600	75	50	7	15	No
7	1400	145	10	20	18	No
8	1100	178	80	17	150	Yes
9	1575	200	70	25	200	Yes
10	1680	230	90	40	48	Yes
11	2250	270	120	60	75	Yes
12	700	120	20	5	10	No
13	1275	50	30	10	17	No
14	350	200	50	22	10	No
15	3310	320	240	80	40	Yes
16	200	120	20	100	12	Yes

classified. It is evident that LOC of a class is the strongest instance in this learning mechanism.

In addition, it can also be analysed from the performed experiments that optimal utmost important metrics are not always selected. Thus, the set of metrics given to the model as an input are re-arranged and some of them are removed. Therefore, the model is able to generate more precise and better decision trees. The process of selecting relevant set of metrics can be automatized by using SPOA in conjunction with proposed approach.

3.3 Proposed Hybrid Approach

Most of the metaheuristic algorithms are based on the collective behavior of social creatures. The collective intelligence is inspired by the interaction of swarm with

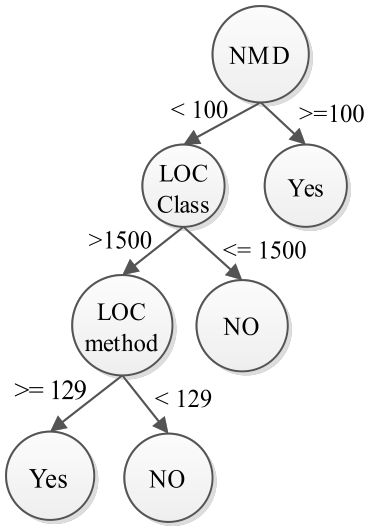


Fig. 3.3. An example of set of instances for blob code smell and derived decision tree.

each other and their environment. Generally, these are easier to implement than evolutionary based algorithms due to fewer operators (i.e., selection, crossover, and mutation). The main components of metaheuristic algorithms are exploration and exploitation [141, 142]. Exploration ensures the algorithm to reach different promising regions of the search space, whereas exploitation ensures the searching of optimal solutions within the promising region [143]. The fine tuning of these components is required to achieve the optimal solution for a given problem. Therefore, an optimizer cab be used to address such type of problems.

3.3.1 Optimization

Many real-life problems need to achieve several objectives such as minimize risks, minimize cost, maximize reliability, and so on [144, 145, 146]. The main aim of optimization is to find the best optimal solution which is to be minimized or maximized [147, 148]. The mathematical formulation of optimization problem is as follows:

$$\text{Minimize/Maximize: } F(\vec{z}) = f_1(\vec{z}) \tag{3.1}$$

$$\text{Subject to:} \quad g_j(\vec{z}) \geq 0, j = 1, 2, \ldots, p \tag{3.2}$$

$$h_j(\vec{z}) = 0, j = 1, 2, \ldots, q \tag{3.3}$$

$$lb_j \leq z_j \leq ub_j, j = 1, 2, \ldots, r \tag{3.4}$$

where p is the number of inequality constraints, g_j is j^{th} inequality constraints, q is the number of equality constraints, h_j is j^{th} equality constraints, r is the number of variables, lb_j and ub_j are lower and upper bounds of j^{th} variable, respectively.

3.3.2 Metaheuristic Techniques

Metaheuristics are broadly classified into two categories namely, single solution and population based algorithms. Single solution based approaches are in which a solution is randomly generated and improved until the best result is obtained. In Population based algorithms, a set of solutions is randomly generated in the given search space and solution values are updated during the course of iterations until the best solution is found [149, 150].

However, single solution based algorithms may trap into local optima. It reforms only one solution which is randomly generated for a given problem. On the other hand, population based algorithms are able to find the global optimum. Due to this, researchers have attracted towards population based algorithms nowadays.

The categorization of population based metaheuristic algorithms is based on the theory of evolutionary, logical behavior of physics, swarm intelligence, and biological behaviors [151, 152, 153] as shown in Fig. 3.4. Evolutionary algorithms are inspired by the evolutionary processes such as reproduction, mutation, recombination, and selection. These algorithms are based on the survival fitness of candidate in a population (i.e., a set of solutions). The physics law based algorithms are inspired by physical processes according to some physics rules such as gravitational force, electromagnetic force, inertia force, heating and cooling of materials. Swarm intelligence based algorithms are inspired by the collective intelligence of swarms.

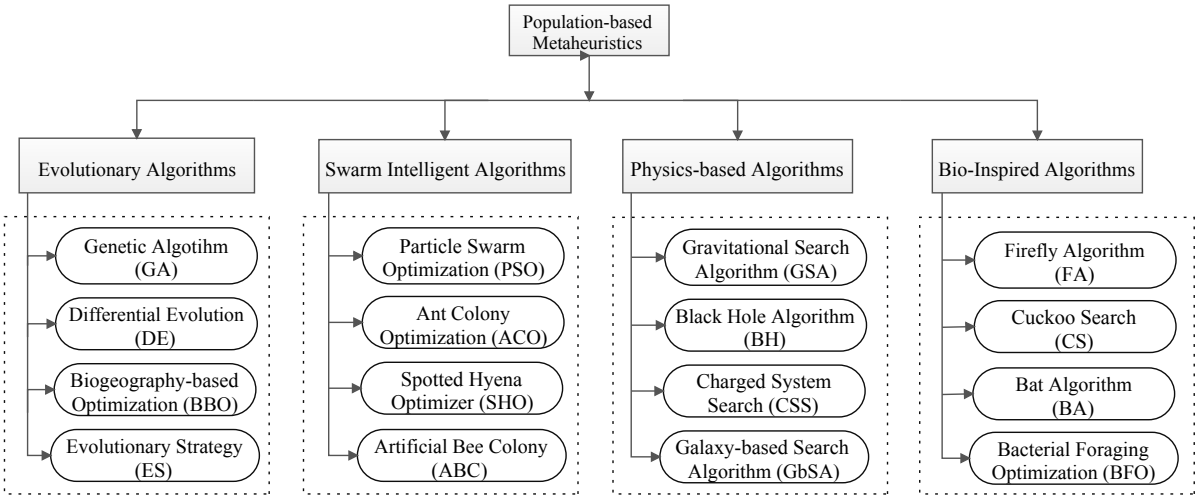


Fig. 3.4. Classification of metaheuristic techniques.

Generally, swarm intelligence based algorithms are very easy to implement than the other algorithms due to less number of parameters are required. The well-known swarm intelligence technique is Particle Swarm Optimization (PSO) [154]. PSO is inspired by the social behavior of fish schooling or bird flocking. Each particle can move around the search space and update its current position with respect to the global best solution.

One of the biggest problem in machine-learning approaches is its parameters values. However, optimization algorithm plays an important role for the selection of appropriate values of parameters and chooses the best values for the problem. Hence, there is a need to optimize the parameters of these approaches.
The main motivation of this work is to develop a novel metaheuristic algorithm and hybridized it with machine-learning approach for solving software engineering problems.

3.3.3 Sandpiper Optimization Algorithm (SPOA)

This section discusses the motivation and computational modeling of the proposed algorithm.

3.3.3.1 Biological Paradigm

Sandpipers are seabirds, which can be found all over the planet. There is the wide range of sandpipers species with different masses and lengths. Sandpipers are omnivorous, and eat insects, fish, reptiles, amphibians, earthworms, and so on. Sandpipers are very sharp birds and they generally live in colonies. They use their intelligence to find and attack the prey.

The most important thing about the sandpipers is their searching and attacking behaviors. Searching is defined as the seasonal movement of sandpipers from one place to another to locate the food rich and abundant sources that will provide required energy. Sandpipers frequently attack migrating birds over the sea, when they migrate from one place to another. They can make their natural round shape movement during attacking.

These behaviors can be formulated in such a way that it can be associated with the objective function to be optimized. This makes it possible to formulate a new optimization algorithm. This approach focuses on two natural behaviors of sandpipers namely migration and attacking.

3.3.3.2 Mathematical Model

The mathematical models of searching and attacking the prey are discussed.

- **Searching (*Intensification*)** During migration, the algorithm simulates how the group of sandpipers move from one position to another.

$$\vec{S} = \vec{M} \times \vec{P}_s(x) \tag{3.5}$$

where $\vec{S}$ represents the position of search agent, which does not collide with other search agent, $\vec{P}_s$ represents the current position of search agent, x signify the present iteration, and $\vec{M}$ represents the movement behavior of

search agent in a given search space.

$$\begin{aligned} &\vec{M} = f - (x \times (f/Max_{iteration})) \\ &\text{where: } x = 0, 1, 2, \ldots, Max_{iteration} \\ &\vec{C} = B \times (\vec{P_{best}}(x) - \vec{P_s}(x)) \end{aligned} \tag{3.6}$$

where f is introduced to control the frequency of employing vector $\vec{M}$, which is linearly decreased from f to 0. In this work, the value of f is set to 2. Vector $\vec{C}$ represents the positions of search agent $\vec{P_s}$ towards the best fit search agent $\vec{P_{best}}$. The behavior of B is randomized which is responsible for proper balancing between intensification and diversification. The B is calculated, as:

$$\begin{aligned} &B = 2 \times \vec{M} \times Rand() \\ &\vec{D} = \mid \vec{S} + \vec{C} \mid \end{aligned} \tag{3.7}$$

where $\vec{D}$ represents the distance between the search agent. $Rand()$ is a random number lies in the range of $[0, 1]$.

- **Attacking (*Diversification*)** The diversification intends to exploit the history and experience of the search process. Sandpipers can continuously change the angle of attack as well as their speed. While attacking the prey, the round behavior occurs in the air. This behavior is described as follows:

$$R = r_d \times cos(i) \times sin(i) \tag{3.8}$$

$$r_d = s \times e^{it} \tag{3.9}$$

where r_d is the radius of each turn of the round, i is a random number in range $[0 \leq i \leq 2\pi]$. s and t are constants to define the round shape, and e is the base of the natural logarithm. The updated position of search agent is calculated as:

$$\vec{P_s}(x) = (\vec{D} \times R) + \vec{P_{best}}(x) \tag{3.10}$$

where $\vec{P_s}(x)$ stores the best solution and update each search agent position.

Algorithm 3 Sandpiper Optimization Algorithm

Input: the sandpipers population $\vec{P_s}$
Output: the leading search agent, $\vec{P_{best}}$
procedure SPOA
Compute fitness for each search agent
$\vec{P_{best}} \leftarrow$ the leading search agent
  while $(x < Max_{iteration})$ **do**
    for each search agent **do**
      Position updation for current agent by Eq. (3.10)
    end for
    Inspect if any search agent crosses the search space and if so then readjust
    Compute the fitness value of each search agent
    Update $\vec{P_{best}}$ if there is a superior solution than already present optimal solution
    $x \leftarrow x + 1$
  end while
return $\vec{P_{best}}$
end procedure

In the first step, SPOA algorithm population is generated randomly. Let us take an example, if the search agent value is 5 then the population is represented as given in Table 3.2. In the second step, the fitness value of these search agents is calculated on benchmark test function. It has been observed that the fitness value of search agent labeled as number 1 is less as compared to others. In this example, Sphere test function [155] is used as objective function which is mathematically described as follows:

$$\begin{aligned} F_1(z) &= \sum_{i=1}^{30} z_i^2 \\ -100 &\leq z_i \leq 100, \\ f_{min} &= 0, \\ Dim &= 30 \end{aligned} \tag{3.11}$$

In the third step, first row is chosen from Table 3.2 and apply the SPOA algorithm operations on it.

Now, compute the attacking behavior of SPOA algorithm.

Table 3.2: Population representation (*Ist Iteration*).

Search agents	1	2	3	4	5	Objective value
1	126.1	297.2	285	385.12	750.2	**1.002**
2	410.2	109.5	100	315.2	350.3	4.121
3	123.7	348.6	340	131.4	182.9	2.025
4	425.1	359.4	754	652.7	472.2	1.345
5	125.7	231.2	365.1	479.1	521.6	1.239

Table 3.3: Updated positions of search agents (*Ist Iteration*).

Search agents	1	2	3	4	5
1	126.1	297.2	285	385.12	750.2
2	410.2*B	109.5*B	100*B	315.2*B	350.3*B
3	123.7*B	348.6*B	340*B	131.4*B	182.9*B
4	425.1*B	359.4*B	754*B	652.7*B	472.2*B
5	125.7*B	231.2*B	365.1*B	479.1*B	521.6*B

In the fourth step, compute the value of variable f. If the value of x is 0 and maximum number of iterations is 1000, then the value of f is 2, i.e., (2 - 0$\times$(2/1000)) = 2, and so on for further steps. Hence, the updated values are obtained as shown in Table 3.3.

In the fifth step, again calculate the values. For example, in Table , the optimal value is 1.002. Now the updated optimal value is 1.629.

In the sixth step, calculate the value of updated search agent w.r.t. above-mentioned calculated values. Repeat this whole process for second iteration as given in Table 3.4. The best search agent is 4 whose objective value is 1.156. Again compute the values of updated search agents as shown in Table 3.5.

Finally, in the seventh and last step, the updated positions are returned by SPOA

Table 3.4: Population representation (*IInd Iteration*).

Search agents	1	2	3	4	5	Objective value
1	147.2	215.7	359	305.4	975.2	2.025
2	402.7	110.1	95	310.7	648.2	2.326
3	127.7	145.3	457	567.8	734.2	1.578
4	449.4	456.3	114	342.7	450.2	**1.156**
5	356.6	645.3	636.1	453.7	784.3	5.110

Table 3.5: Updated positions of search agents (*IInd Iteration*).

Search agents	1	2	3	4	5
1	147.2*B	215.7*B	359*B	305.4*B	975.2*B
2	402.7*B	110.1*B	95*B	310.7*B	648.2*B
3	127.7*B	145.3*B	457*B	567.8*B	734.2*B
4	449.4	456.3	114	342.7	450.2
5	356.6*B	645.3*B	636.1*B	453.7*B	784.3*B

algorithm and stops the algorithm once it reached the maximum number of iterations.

3.3.3.3 Computational Complexity

The complexity of an algorithm is an important metric to judge its performance. The population initialization process of the proposed SPOA and other competing algorithms requires $\mathcal{O}(n_o \times n_p)$ time, where n_o depicts the count of objectives and n_p depicts the count of population size. The complexity of calculating the fitness of search agents for all algorithms needs $\mathcal{O}(Max_{iteration} \times o_f)$, where o_f represents the objective function for a given problem.

To simulate the whole procedure, it requires $\mathcal{O}(N)$ time. On the other hand, the space complexity of an algorithm is the maximum utilized space at any instant of time after the algorithm is being initialized. In this work, the space complexity for proposed approach is considered, as $\mathcal{O}(n_o \times n_p)$.

The computational complexity of the proposed SPOA algorithm is $\mathcal{O}(N \times n_o \times n_p \times o_f \times Max_{iteration})$.

3.3.3.4 Results and Discussions

This section contains the experimental validation of the proposed approach using 23 widely used benchmark test functions followed by performance comparison with other competitor methods. These test functions are categorised into three categories, such as unimodal, multimodal, and fixed-dimension multimodal [155, 156]. The description of these test functions is given in Appendix A.

Further, for doing the comparison of the proposed algorithm, nine existing well-known metaheuristics are selected, which include Spotted Hyena Optimizer (SHO) [155], Grey Wolf Optimizer (GWO) [157], Particle Swarm Optimization (PSO) [154], Moth-Flame Optimization (MFO) [158], Multi-Verse Optimizer (MVO) [159], Sine Cosine Algorithm (SCA) [160], Gravitational Search Algorithm (GSA) [161], Genetic Algorithm (GA) [162], and Differential Evolution (DE) [163].

The parameter settings of the SPOA and other competitor algorithms are given in Table 3.6. The experimental evaluation and algorithms are implemented in MATLAB R2017a (version 9.2), and simulations are performed in the environment of Microsoft Windows 10 with 64 bit on Core i7 processor with 3.2 GHz and 16 GB memory. The proposed algorithm utilizes 30 independent runs to generate the obtained results. The best results are in bold form and the italic results represent the standard deviation.

Table 3.6: Parameter setting values for algorithms involved.

Algorithms	Parameters	Values
Spotted Hyena Optimizer (SHO)	$\vec{h}$ $\vec{M}$	[5, 0] [0.5, 1]
Grey Wolf Optimizer (GWO)	$\vec{a}$	[2, 0]
Particle Swarm Optimization (PSO)	Inertia Coefficient Cognitive and Social Coeff	0.75 1.8, 2
Moth-Flame Optimization (MFO)	Convergence Constant Logarithmic Spiral	[-1, -2] 0.75
Multi-Verse Optimizer (MVO)	Wormhole Exis. Probability Travelling Distance Rate	[0.2, 1] [0.6, 1]
Sine Cosine Algorithm (SCA)	Number of Elites	2
Gravitational Search Algorithm (GSA)	Gravitational Constant Alpha Coefficient	100 20
Genetic Algorithm (GA)	Crossover Mutation	0.9 0.05
Differential Evolution (DE)	Crossover Scale factor (F)	0.9 0.5

3.3.3.5 Diversification Analysis

The unimodal test functions are suitable to assess and evaluate the diversification capability of the algorithm. Table 3.7 shows the optimal values obtained by the proposed and above-mentioned algorithms for unimodal benchmark test functions. It can be seen that SPOA and SHO algorithms provide better results on $F_1(Sphere)$ test function. The algorithm GWO and GSA rank second and third, respectively, in context to average and standard deviation. For $F_2(Schwefel's\,Problem\,2.22)$ test function, the SPOA and SHO algorithms are significantly better than other competitor algorithms. After these, the GWO and GA are the second and third best algorithms for this benchmark test function, respectively. The SHO algorithm obtains the best results on F_3 $(Schwefel's\,Problem\,1.2)$ test function, whereas the SPOA and GWO algorithms become second and third best optimizer, respectively.

For $F_4(Schwefel's\,Problem\,2.21)$ test function, the GWO obtains better results than the other algorithms. The SHO is the second best and SPOA is the third best optimizer on this benchmark test function. The results obtained by the SPOA algorithm are better than the other approaches on $F_5(Generalized\,Rosenbrock's)$ test function. For F_5 test function, the SHO and GWO are the second and third best optimization algorithm. The GSA algorithm obtains competitive results than the other metaheuristics on $F_6(Step)$ test function. On this benchmark test function, the PSO and MFO algorithms are the second and third best optimization algorithms.

For $F_7(Quartic)$ test function, the results obtained by SPOA algorithm are superior to others. On the other hand, the SHO and GA algorithms outperform over other algorithms as a second and third best algorithm, respectively. Results state that the proposed SPOA algorithm identifies the best results corresponding to most of the test functions. In particular, the SPOA reveals the robustness for functions $F_1, F_2, F_5,$ and F_7, which show the capability for diversification around

the optimum point.

3.3.3.6 Intensification Analysis

These functions have a capability to avoid local optima problem. Multimodal and fixed-dimension multimodal test functions are suitable for better intensification of a proposed algorithm. Tables 3.8 and 3.9 depicts the results of different methods on multimodal and fixed-dimension multimodal benchmark test functions, respectively.

For F_8 $(Generalized\,Schwefel's\,Problem\,2.26)$ function, the SPOA algorithm outperforms than the existing competitive approaches. The MFO and MVO are the second and third best optimization algorithms, respectively. For $F_9(Generalized\,Rastrigin's\,Function)$ function, SHO becomes the best optimizer than others. The GWO and GA are the second and third best optimization algorithms on F_9 test function. For $F_{10}(Ackley's\,Function)$ function, the average value of the SPOA algorithm is the best as compared to other approaches, whereas the standard deviation of the GWO algorithm is better than the other metaheuristics. For $F_{11}(Generalized\,Griewank\,Function)$ function, the performance of the SPOA and SHO algorithms are same, whereas the performance of the GA and DE algorithms are the second and third best than competitive optimizers.

The results obtained by the PSO algorithm are superior to other metaheuristics on $F_{12}(Generalized\,Penalized\,Function)$ function. For $F_{13}(Generalized\,Penalized\,Function)$ function, the GSA is the best optimization algorithm, while the PSO and GA obtain better results than the others. For $F_{14}(Shekel's\,Foxholes\,Function)$ function, the MVO provides better results. The SCA and MFO are second and third best optimization algorithms, respectively. For $F_{15}(Kowalik's\,Function)$ and $F_{16}(Six-HumpCamel-Back\,Function)$ functions, the results of the SPOA are superior than the other algorithms. The DE and SHO algorithms are the second and third best optimizers, respectively. For

Table 3.7: Results of unimodal benchmark functions.

F	SPOA	SHO	GWO	PSO	MFO	MVO	SCA	GSA	GA	DE
F_1	**0.00E+00**	0.00E+00	4.69E-47	4.98E-09	3.15E-04	2.81E-01	3.55E-02	1.16E-16	1.95E-12	3.10E-10
	(0.00E+00)	*(0.00E+00)*	*(4.50E-48)*	*(1.41E-10)*	*(5.00E-05)*	*(1.10E-02)*	*(1.07E-02)*	*(6.10E-17)*	*(2.04E-13)*	*(3.21E-11)*
F_2	**0.00E+00**	0.00E+00	1.20E-24	7.29E-04	3.71E+01	3.96E-01	3.23E-05	1.70E-01	6.53E-18	4.99E-13
	(0.00E+00)	*(0.00E+00)*	*(1.00E-24)*	*(1.57E-05)*	*(2.16E+01)*	*(1.41E-01)*	*(8.28E-06)*	*(5.29E-02)*	*(4.00E-19)*	*(4.81E-14)*
F_3	4.62E-19	**0.00E+00**	1.00E-14	1.40E+01	4.42E+03	4.31E+01	4.91E+03	4.16E+02	7.70E-10	5.58E-02
	(2.30E-27)	***(0.00E+00)***	*(4.18E-15)*	*(7.13E+00)*	*(3.71E+03)*	*(8.97E+00)*	*(3.89E+03)*	*(1.56E+02)*	*(5.46E-12)*	*(1.50E-03)*
F_4	7.35E-05	7.78E-12	**2.02E-14**	6.00E-01	6.70E+01	8.80E-01	1.87E+01	1.12E+00	9.17E+01	5.21E+01
	(3.11E-06)	*(5.06E-12)*	***(2.95E-15)***	*(1.72E-01)*	*(1.06E+01)*	*(2.50E-01)*	*(8.21E+00)*	*(9.89E-01)*	*(5.67E+01)*	*(1.01E-01)*
F_5	**7.00E+00**	8.59E+00	2.79E+01	4.93E+01	3.50E+03	1.18E+02	7.37E+02	3.85E+01	5.57E+02	2.10E+02
	(1.13E-01)	*(5.53E-01)*	*(1.84E+00)*	*(3.89E+01)*	*(2.83E+03)*	*(1.00E+02)*	*(1.90E+02)*	*(3.47E+01)*	*(4.16E+01)*	*(1.73E+02)*
F_6	3.47E-02	2.46E-01	6.58E-01	9.23E-09	1.66E-04	3.15E-01	4.88E+00	**1.08E-16**	3.15E-01	9.77E-02
	(1.31E-04)	*(1.78E-01)*	*(3.38E-01)*	*(8.48E-10)*	*(2.49E-05)*	*(9.98E-02)*	*(9.75E-01)*	***(4.00E-17)***	*(9.98E-02)*	*(5.13E-02)*
F_7	**3.35E-06**	3.29E-05	7.80E-04	6.92E-02	3.22E-01	2.02E-02	3.88E-02	7.68E-01	6.79E-04	4.00E-03
	(7.51E-06)	*(2.43E-05)*	*(3.85E-04)*	*(2.87E-02)*	*(2.93E-01)*	*(7.43E-03)*	*(1.15E-02)*	*(5.70E-01)*	*(5.21E-04)*	*(2.27E-03)*

$F_{17}(Branin\,Function)$ and $F_{18}(Goldstein-Price\,Function)$ functions, the results produced from the SPOA algorithm are better than others.

For $F_{19}(Hartman's)$ function, the average value obtained by the PSO algorithm is better, while the standard deviation value obtained by the MFO algorithm is better than competitive approaches. For $F_{20}(Hartman's)$ function, the SPOA algorithm performs superior, whereas the PSO and GWO are the second and third best optimizers, respectively. The average value of the GWO algorithm is the best on $F_{21}(Shekel's\,Foxholes\,Function)$ test function, while the standard deviation value of the SHO algorithm is better on this test instance. The results obtained by the SPOA algorithm are superior to the competitor approaches on $F_{22}(Shekel's\,Foxholes\,Function)$ and $F_{23}(Shekel's\,Foxholes\,Function)$ benchmark test functions. Results state that the SPOA algorithm has good intensification capability and competitive on seven test problems (i.e., $F15, F16, F17, F18, F20, F22,$ and $F23$) out of ten test functions.

3.3.3.7 Analysis of SPOA Algorithm

The convergence analysis of metaheuristic algorithm is an another feature for the better understanding of intensification and diversification mechanisms. Sandpipers tend to explore the varying potential regions within a search space.

Table 3.8: Results of multimodal benchmark functions.

F	SPOA	SHO	GWO	PSO	MFO	MVO	SCA	GSA	GA	DE
F_8	**-8.50E+03**	-1.16E+03	-6.14E+03	-6.01E+03	-8.04E+03	-6.92E+03	-3.81E+03	-2.75E+03	-5.11E+03	-3.94E+03
	(4.57E+02)	***(2.72E+02)***	*(9.32E+02)*	*(1.30E+03)*	*(8.80E+02)*	*(9.19E+02)*	*(2.83E+02)*	*(5.72E+02)*	*(4.37E+02)*	*(5.81E+02)*
F_9	3.12E+02	**0.00E+00**	4.34E-01	4.72E+01	1.63E+02	1.01E+02	2.23E+01	3.35E+01	1.23E+01	8.50E+01
	(2.85E+01)	***(0.00E+00)***	*(2.68E-01)*	*(1.03E+01)*	*(3.74E+01)*	*(1.89E+01)*	*(1.28E+01)*	*(1.19E+01)*	*(1.01E+01)*	*(1.00E+01)*
F_{10}	**4.22E-16**	2.48E+00	1.63E-14	3.86E-02	1.60E+01	1.15E+00	1.55E+01	8.25E-09	5.31E-11	7.40E-06
	(4.09E-13)	*(1.41E+00)*	***(3.14E-15)***	*(2.91E-04)*	*(6.18E+00)*	*(7.87E-01)*	*(8.11E+00)*	*(1.90E-09)*	*(9.10E-12)*	*(5.44E-07)*
F_{11}	**0.00E+00**	0.00E+00	2.29E-03	5.50E-03	5.03E-02	5.74E-01	3.01E-01	8.19E+00	3.31E-06	7.09E-04
	(0.00E+00)	*(0.00E+00)*	*(1.21E-03)*	*(4.32E-04)*	*(1.29E-02)*	*(1.12E-01)*	*(2.89E-01)*	*(3.70E+00)*	*(1.43E-06)*	*(5.58E-04)*
F_{12}	5.80E-01	3.68E-02	3.93E-02	**1.05E-10**	1.26E+00	1.27E+00	5.21E+01	2.65E-01	9.16E-08	1.06E-06
	(4.71E-02)	*(1.15E-02)*	*(2.42E-02)*	***(2.86E-11)***	*(1.83E+00)*	*(1.02E+00)*	*(2.19E+01)*	*(1.13E-01)*	*(2.80E-08)*	*(1.00E-06)*
F_{13}	8.48E-02	9.29E-01	4.75E-01	4.03E-03	7.24E-01	6.60E-02	2.81E+02	**5.73E-32**	6.39E-02	9.78E-02
	(4.32E-04)	*(9.52E-02)*	*(2.38E-01)*	*(3.01E-03)*	*(1.59E-01)*	*(4.33E-02)*	*(5.62E+01)*	***(1.30E-32)***	*(4.49E-02)*	*(3.42E-02)*

In the first step of optimization process, the search agents can change rapidly. Figs. 3.5 to 3.6 shows the convergence curves of the SPOA and other competitor methods. The SPOA shows three different convergence behaviors while optimizing test functions.

Initially, the SPOA converges quickly towards the potential regions because of its adaptive mechanism. This behavior is shown in $F1, F3, F7, F9$ and $F11$ test functions. The GSA encourages good convergence and achieves better performance on $F5$ test function during the initial steps of iterations. On the other hand, the MFO converges towards the optimum for $F21$ and $F23$ test functions. In the second step, the SPOA converges towards the optimum during the final iteration, which is observed in $F21$ and $F23$ test functions. The GA and MFO algorithms also converge on $F1$, $F3$ and $F11$ test functions, the SCA and MVO on $F5$ test function, and the PSO algorithm on $F7$ and $F9$ test functions towards the optimum during final iterations. The last and third step is the explicit convergence. This behavior is shown for the SPOA algorithm on $F21$ and $F23$ test functions, for the MFO, GA, and DE on $F9$ test function.

3.3.3.8 Scalability Study

The proposed algorithm is implemented on highly scalable environment to solve the large-scale optimization problems. The dimensionality of the various test

Table 3.9: Results of fixed-dimension multimodal benchmark functions.

F	SPOA	SHO	GWO	PSO	MFO	MVO	SCA	GSA	GA	DE
F_{14}	3.35E+00	9.68E+00	3.71E+00	2.77E+00	2.21E+00	**9.98E-01**	1.26E+00	3.61E+00	4.39E+00	3.97E+00
	(1.15E+00)	*(3.29E+00)*	*(2.15E+00)*	*(2.20E+00)*	*(1.80E+00)*	***(4.13E-02)***	*(6.86E-01)*	*(2.96E+00)*	*(4.41E-02)*	*(1.42E+00)*
F_{15}	**4.11E-04**	9.01E-04	3.66E-03	9.09E-04	1.58E-03	7.15E-03	1.01E-03	6.84E-03	7.36E-03	5.01E-04
	(4.26E-05)	*(1.06E-04)*	*(3.48E-04)*	*(2.38E-04)*	*(2.48E-04)*	*(1.40E-03)*	*(3.75E-04)*	*(2.77E-03)*	*(2.39E-04)*	*(1.00E-04)*
F_{16}	**-1.08E+01**	-1.06E+01	-1.03E+00	-1.03E+00	-1.03E+00	-1.03E+00	-1.03E+00	-1.03E+00	-1.04E+00	-1.03E+00
	(6.48E-12)	*(2.86E-11)*	*(7.02E-09)*	*(1.00E+00)*	*(1.01E+00)*	*(4.74E-08)*	*(3.23E-05)*	*(0.00E+00)*	*(4.19E-07)*	*(3.60E-04)*
F_{17}	**3.98E-01**	3.98E-01	3.98E-01	3.98E-01	3.98E-01	3.98E-01	3.99E-01	3.98E-01	3.98E-01	3.98E-01
	(1.36E-03)	*(2.46E-01)*	*(2.42E-02)*	*(4.11E-02)*	*(1.13E-03)*	*(2.15E-02)*	*(7.61E-04)*	*(1.74E-04)*	***(1.00E-04)***	*(8.40E-02)*
F_{18}	**3.00E+00**	3.00E+00	3.00E+00	3.00E+00	3.00E+00	5.70E+00	3.00E+00	3.01E+00	3.01E+00	3.00E+00
	(5.49E-05)	*(8.15E-05)*	*(7.16E-06)*	*(6.59E-05)*	***(1.42E-07)***	*(8.40E-04)*	*(2.25E-05)*	*(3.24E-02)*	*(6.33E-07)*	*(9.68E-04)*
F_{19}	-3.88E+00	-3.75E+00	-3.86E+00	**-3.90E+00**	-3.86E+00	-3.86E+00	-3.86E+00	-3.22E+00	-3.30E+00	-3.40E+00
	(3.00E-10)	*(4.39E-01)*	*(1.57E-03)*	*(3.37E-15)*	***(3.16E-15)***	*(3.53E-07)*	*(2.55E-03)*	*(4.15E-01)*	*(4.37E-10)*	*(6.56E-06)*
F_{20}	**-3.32E+00**	-1.44E+00	-3.27E+00	-3.32E+00	-3.23E+00	-3.23E+00	-2.84E+00	-1.47E+00	-2.39E+00	-1.79E+00
	(1.23E-02)	*(5.47E-01)*	*(7.27E-02)*	*(2.66E-01)*	*(6.65E-02)*	*(5.37E-02)*	*(3.71E-01)*	*(5.32E-01)*	*(4.37E-01)*	*(7.07E-02)*
F_{21}	-1.00E+01	-1.00E+01	**-1.01E+01**	-1.00E+01	-1.00E+01	-1.00E+01	-1.00E+01	-1.00E+01	-1.00E+01	-1.00E+01
	(3.47E+00)	***(3.80E-01)***	*(1.54E+00)*	*(2.77E+00)*	*(3.52E+00)*	*(2.91E+00)*	*(1.80E+00)*	*(1.30E+00)*	*(2.34E+00)*	*(1.91E+00)*
F_{22}	**-1.04E+01**	-1.04E+01	-1.03E+01	-1.04E+01	-1.04E+01	-1.01E+01	-1.04E+01	-1.04E+01	-1.04E+01	-1.04E+01
	(2.00E-04)	*(2.04E-04)*	*(2.73E-04)*	*(3.08E+00)*	*(3.20E+00)*	*(3.02E+00)*	*(1.99E+00)*	*(2.64E+00)*	*(1.37E-02)*	*(5.60E-03)*
F_{23}	**-1.05E+01**	-1.05E+01	-1.01E+01	-1.05E+01	-1.05E+01	-1.03E+01	-1.05E+01	-1.05E+01	-1.05E+01	-1.05E+01
	(1.32E-01)	*(2.64E-01)*	*(8.17E+00)*	*(2.52E+00)*	*(3.68E+00)*	*(3.13E+00)*	*(1.96E+00)*	*(2.75E+00)*	*(2.37E+00)*	*(1.60E+00)*

functions is varied from 30-50, 50-80, and 80-100. Fig. 3.7 depicts that the performance evaluation of the SPOA and shows the robustness of it. It is also observed that the performance degradation is not too much which makes it applicable on the highly scalable environment.

3.3.3.9 Statistical Testing

The comparison of algorithms does not guarantee the efficacy and superiority of the proposed algorithm. The possibility of getting good results, by chance, can not be ignored. For this, Wilcoxon statistical test [164] is performed at 5% level of significance, and the $p-$values are tabulated in Table 3.10. During statistical testing, the best algorithm is chosen and compared with other competitor algorithms. The obtained $p-$values that are less than 0.05 demonstrate the statistical superiority of the proposed SPOA. The test functions are taken from IEEE CEC 2015 special session [165], which are the most challenging test functions available in the literature. From results, it can be concluded that the results of the SPOA are significantly effective than competitive approaches (i.e., SHO, GWO, PSO, MFO, MVO, SCA, GSA, GA, and DE) over IEEE CEC 2015 benchmark test

suite.

Table 3.10: $p-$values obtained from Wilcoxon test for IEEE CEC 2015 benchmark functions.

CEC	SHO	GWO	PSO	MFO	MVO	SCA	GSA	GA	DE
1	0.0005	0.0002	0.0002	0.0003	0.0080	0.0003	0.0001	0.0006	0.0050
2	0.0012	0.0001	0.0002	0.0060	0.0001	0.0051	0.0035	0.0020	0.0050
3	0.0001	0.0200	0.0100	0.0165	0.0001	0.0090	0.0008	0.0121	0.0001
4	0.0061	0.0380	0.0250	0.0051	0.0035	0.0042	0.0008	0.0092	0.0001
5	0.0006	0.0450	0.0001	0.0687	0.0001	0.2410	0.0001	0.0008	0.0498
6	0.0001	0.0004	0.0033	0.0050	0.8000	0.4700	0.0001	0.0046	0.0557
7	0.0090	0.0086	0.0001	0.0033	0.0037	0.0021	0.0001	0.0075	0.0013
8	0.0091	0.7910	0.0001	0.0859	0.0031	0.0001	0.0223	0.0060	0.0087
9	0.0008	0.0011	0.0001	0.0211	0.0001	0.0001	0.0096	0.0190	0.0001
10	0.0001	0.0001	0.0001	0.0807	0.0001	0.0125	0.0001	0.0510	0.0043
11	0.0027	0.0103	0.0002	0.0001	0.0008	0.0490	0.0045	0.0005	0.0110
12	0.0001	0.0091	0.0001	0.0001	0.0008	0.0001	0.0001	0.0010	0.0251
13	0.0001	0.0045	0.0001	0.0001	0.0393	0.0001	0.0030	0.0022	0.0800
14	0.0001	0.0009	0.0045	0.0001	0.0896	0.0001	0.0501	0.0061	0.0031
15	0.0020	0.0017	0.0024	0.0492	0.0054	0.0001	0.0001	0.0031	0.0099

3.4 Code Smell Detection using Proposed Hybrid Approach

The proposed approach is examined for detection of five code smells namely, Blob, Feature Envy, Data Class, Functional Decomposition, and Spaghetti Code in software systems. The definition of these five code smells is given in Chapter 1. These five code-smell types are chosen in this work because these are the utmost frequent, hard to detect, and refactor. The objective of this problem is the accurate detection of code smells when compared to the other existing techniques.

3.4.1 Research Questions

- **RQ1:** What kind of bad smells does it correctly identify?
- **RQ2:** To what extent the proposed technique identifies the bad smells comparatively existing bad smell detection techniques?

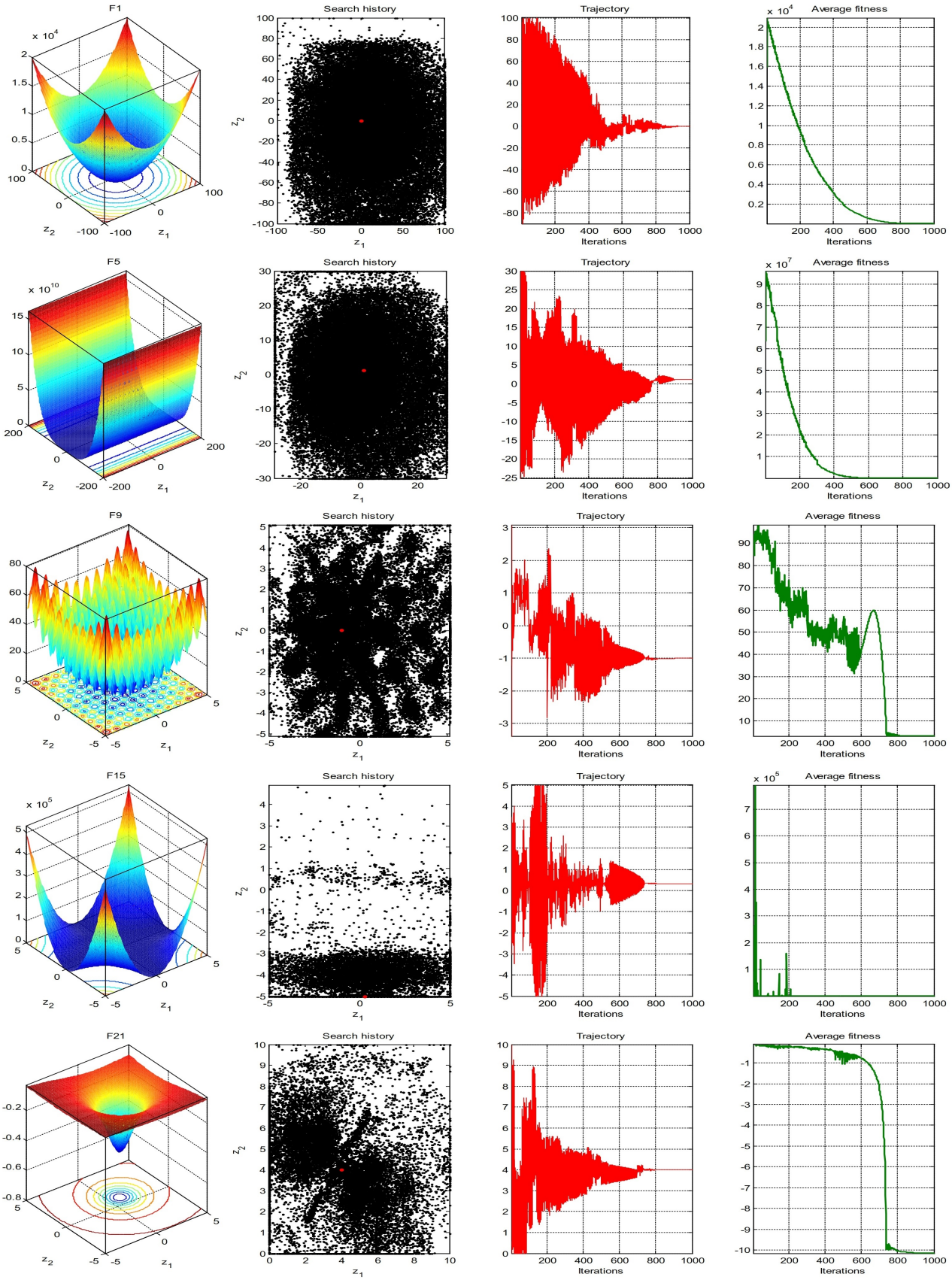


Fig. 3.5. Search history, trajectory, and average fitness of SPOA algorithm on 2D benchmark test problems.

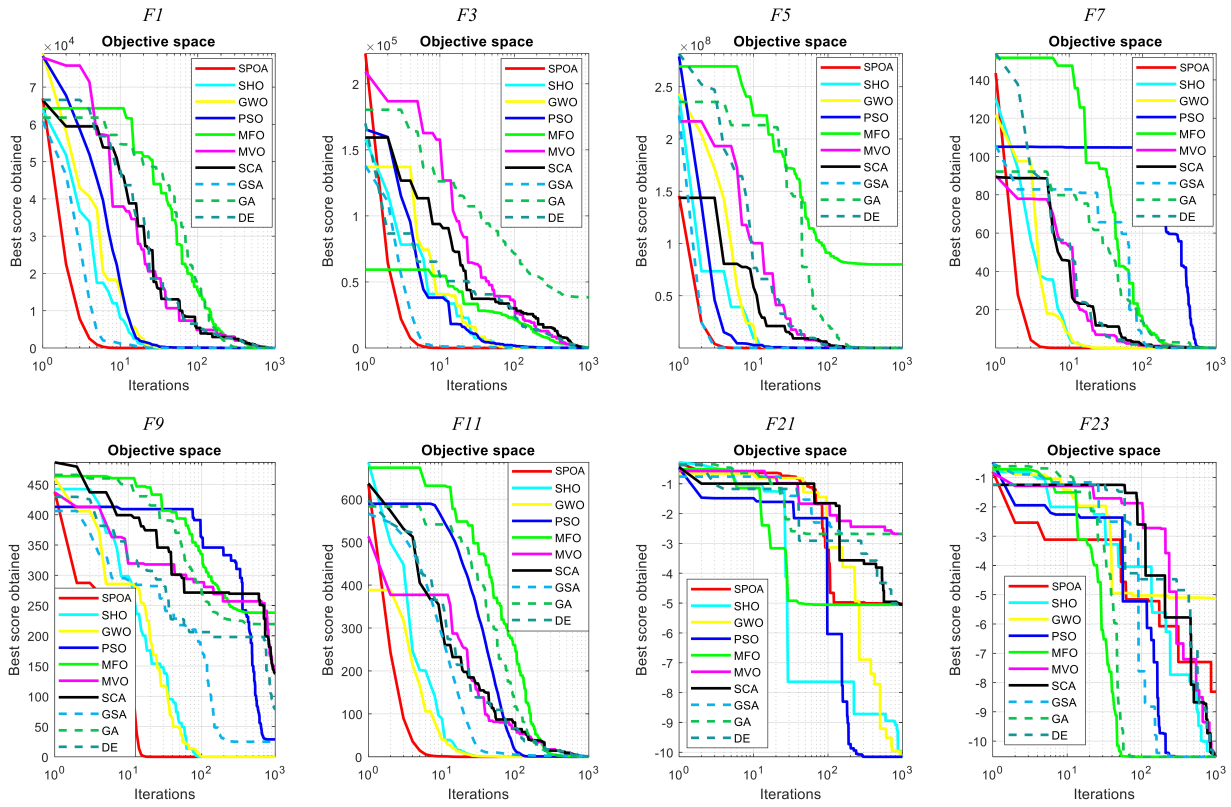


Fig. 3.6. Convergence analysis of proposed SPOA and competitor algorithms on some of the benchmark test functions.

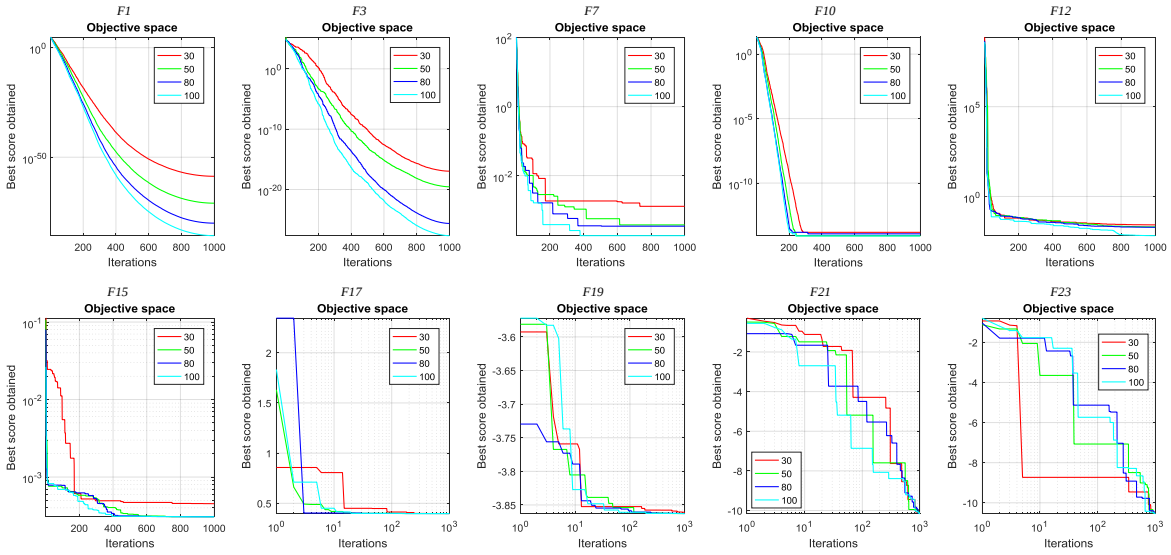


Fig. 3.7. Scalability analysis of proposed SPOA algorithm on some of the benchmark test functions.

To answer RQ1, the authors explored the types of bad smell that are identified by proposed SP-J48 algorithm. Furthermore, in order to answer RQ2, the proposed

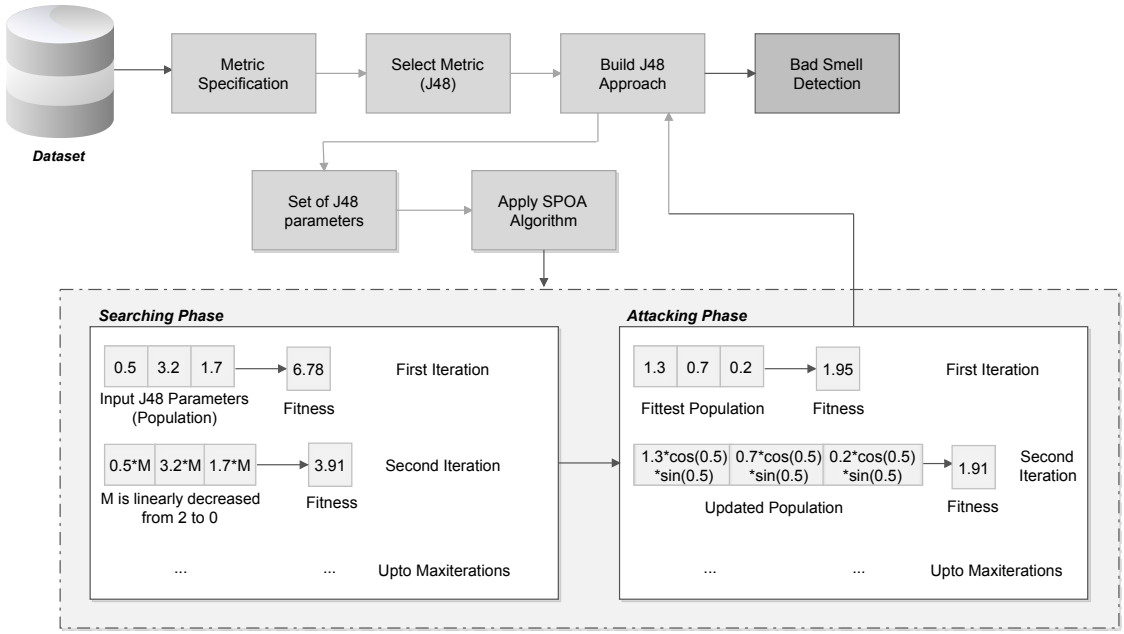


Fig. 3.8. Code smell detection using SP-J48.

SP-J48 algorithm is compared to the existing techniques using four performance metrics viz.: Precision, Recall, and $F_{measure}$.

3.4.2 Datasets or Subject Systems

To validate its performance, the proposed approach is experimented on three Java based open source systems namely, GanttProject, Log4j, and Xerces-J. These datasets are chosen based on different factors: Firstly, they are open source software and are freely available. Secondly, researchers can replicate their study and other researchers who have used the same systems can make comparisons. The detail of these open source software systems is given in Table 3.11.

Table 3.11: Description of considered systems.

S.No.	Project Name	Description	Version	Lines of Code	No. of Classes
1.	GanttProject	Cross-platform for project management and planning	1.10.2	21,267	188
2.	Log4j	Java based logging utility	1.2.1	10,224	189
3.	Xerces-J	Java library for manipulating, validating, and parsing XML documents	2.7.0	71,217	513

Algorithm 4 SP-J48.

Input: Parameters of J48 algorithm, Maximum number of iterations Max_{iter}
Output: Optimal detected code smells

procedure
Choose the initial parameters and normalize it
    $\alpha \leftarrow$ Initial selection of parameters
    $iteration \leftarrow 0$
    while $(iteration < Max_{iter})$ **do**
        $\beta \leftarrow$ Select the best obtained parameters using Algorithm 3
        $iteration \leftarrow iteration + 1$
    end while
    $\gamma \leftarrow$ Set of detected code smells (β)
return γ
end procedure

In this problem, the initial model used are GanttProject and Log4j whereas, Xerces-J is used as base example.

3.4.3 Software Metrics

Software metrics are the measurements of properties specific to fragments of source code of one software, such as, the number of lines of code, number of methods of a class or number of parameters of one method. The metrics of software used in this work are derived from tools CKJM [166] and POM [167], which are capable of calculating the values of the metrics by using the source code of a software given as an input. From the metrics to use, 18 are calculated by the tool CKJM[1] and 44 metrics are calculated by the tool POM[2].

3.4.4 Performance Evaluation

To answer the posed RQ1, the set of computed metrics from both initial model and base example are considered. The overall results obtained by SP-J48 decision tree algorithm in the form of average values of Precision, Recall, and $F_{measure}$ are

[1]WMC, DIT, NOC, CBO, RFC, LCOM, Ca, Ce, NPM, LCOM3, LOC, DAM, MOA, MFA, CAM, IC, CBM and AMC

[2]ACAIC, ACMIC, AID, CLD, DCAEC, DCC, DCMEC, DIT, DSC, ICHClass, IR, LCOM1, LCOM2, LCOM5, LOC, MLOCsum, McCabe, NAD NADExtended, NCM, NMA, NMD, NMDExtended, NMI, NMO, NOA, NOC, NOD, NOF, NOH, NOM, NOP, NOPM, NOParam, NOTC, NOTI, RFC_New, SIX , VGsum, WMC1, WMC_New, cohesionAttributes and connectivity

tabulated in Table 3.15. The obtained findings indicate that in majority of the cases SP-J48 is able to generate effective rules for detecting the code smells from considered software systems.

Moreover, it can also be analysed from Table 3.15 that the $F_{measure}$ value for all the code smells is above 75% except Blob code smell. Hence, it can be concluded that the characteristics of Blob code smell cannot be generalised by the considered approach or it cannot be detected efficiently on the basis of considered software metrics. Likewise, the value of Recall in case of Blob is also less than 45%, which means that less than half of this type of code smells have been detected.

On the other hand, SP-J48 efficiently detected most of the Functional Decomposition, Spaghetti Code, and Feature Envy bad smells as also evident from the values of Precision, Recall, and $F_{measure}$. Additionally, the proposed approach rendered good results for Data Class code smell as well.

Though, the obtained results seem to be promising, but it is hard to draw the conclusion only on the basis of performance metrics. Therefore, to analyze the results from other scenarios these results are also compared with other existing techniques which assists in establishing reliable conclusions.

To answer the posed RQ2, the proposed approach is compared with the existing state-of-art techniques on the basis of performance metrics values. Furthermore, the effectiveness of the proposed approach is analysed by comparing the Precision, Recall, and $F_{measure}$ of the proposed SP-J48 technique with the other techniques. The obtained values for proposed SP-J48 and competitor algorithms (J48 [168], SVM [169], and Bayesian [170]) are tabulated in Tables 3.12 - 3.15 and Figs. 3.9 to 3.11.

In GanttProject, the Precision metric, for finding the Blob defect, of the proposed SP-J48 is less than SVM but better than Bayesian and J48 approach. Whereas, the performance of SP-J48 in terms of Recall and $F_{measure}$ metrics is better than all the competitor approaches for finding the Blob defect. The search ability of SP-J48 is

similar to J48 and better than Bayesian and SVM approach in terms of Precision for Functional decomposition defect. Similarly, for Recall and $F_{measure}$, proposed SP-J48 is similar to J48 and better than all the competitor approaches.

Furthermore, the proposed SP-J48 algorithm is better than all the approaches in terms of Precision, Recall, and $F_{measure}$ performance metrics for finding the Feature envy defect in GanttProject. The Data class defect is easily detected in optimal time as compared to J48 and Bayesian in terms of Precision. Similarly, the Recall and $F_{measure}$ performance metrics are better while employing SP-J48 algorithm. For Spaghetti code defect in GanttProject, the performance of SP-J48 is similar to J48 in terms of Precision and better than others. Likewise, in terms of Recall and $F_{measure}$, the proposed SP-J48 performance is superior than competitive approaches.

In Log4j, the Precision metrics for finding the Blob defect of the proposed SP-J48 is very much similar to SVM, and better than J48 and Bayesian approach. Whereas, the Recall metric of SP-J48 is better than all the competitor approaches for finding the Blob defect. The search ability of SP-J48 is better than J48, SVM, and Bayesian approaches in terms of Precision, Recall, and $F_{measure}$ for Functional decomposition defect.

Further to that, the proposed SP-J48 algorithm is similar to J48 and better than all other approaches in terms of Precision, Recall, and $F_{measure}$ for finding the Feature envy defect in Log4j. The Data class defect is also easily detected by SP-J48 in optimal time as compared to all the algorithms in terms of Precision, Recall, and $F_{measure}$ metrics. For Spaghetti code defect in Log4j, it is analysed that SP-J48 is better than others in terms of Precision. While, in terms of Recall and $F_{measure}$, the proposed SP-J48 again compete all the approaches and show its superiority.

In Xerces-J, the Precision and $F_{measure}$ metrics, for finding the Blob defect, is very much similar to SVM and better than J48 as compared to Bayesian approach. The search ability of SP-J48 is better than J48, SVM, and Bayesian in terms of

Precision for Functional decomposition defect. However, SP-J48 in terms of Recall and $F_{measure}$, is similar to J48 and better than all the other competitor approaches. Similarly, the proposed SP-J48 algorithm is better than all approaches in terms of Precision, Recall, and $F_{measure}$ performance metrics for finding the Feature envy defect in Xerces-J.

The Data class defect is easily detected by SP-J48 and J48 in optimal time as compared to all the algorithms in terms of Precision. Whereas, the Recall and $F_{measure}$ performance metrics of SP-J48 are better than all other considered approaches. The analysis of Spaghetti code defect in Xerces-J in terms of Precision is better than its competitors. It is also analysed that SP-J48 is better than others in terms of Recall and $F_{measure}$.

Table 3.12: Obtained results by J48 approach.

Open source softwares	Defects	Precision	Recall	$F_{measure}$
GanttProject	Blob	65.24	43.25	52.02
	Functional Decomposition	93.65	90.74	92.17
	Spaghetti Code	91.96	89.15	90.53
	Feature Envy	94.65	94.92	94.78
	Data Class	82.82	74.17	78.26
Log4j	Blob	70.54	42.45	53.00
	Functional Decomposition	90.54	88.47	89.49
	Spaghetti Code	92.89	91.53	92.20
	Feature Envy	86.65	87.81	87.23
	Data Class	83.72	72.47	77.69
Xerces-J	Blob	70.54	42.25	52.85
	Functional Decomposition	89.24	87.74	88.48
	Spaghetti Code	91.59	87.83	89.67
	Feature Envy	92.54	94.89	93.70
	Data Class	82.86	71.74	76.90

Table 3.13: Obtained results by SVM approach.

Open source softwares	Defects	Precision	Recall	$F_{measure}$
GanttProject	Blob	71.24	36.25	48.05
	Functional Decomposition	76.65	84.74	80.49
	Spaghetti Code	83.96	59.15	69.40
	Feature Envy	75.65	36.92	49.62
	Data Class	80.82	54.27	64.94
Log4j	Blob	75.49	38.56	51.05
	Functional Decomposition	72.57	85.74	78.61
	Spaghetti Code	89.24	56.38	69.10
	Feature Envy	78.55	48.81	60.21
	Data Class	79.72	59.74	68.30
Xerces-J	Blob	75.45	37.45	50.05
	Functional Decomposition	75.26	84.65	79.68
	Spaghetti Code	84.95	54.80	66.62
	Feature Envy	75.54	37.89	50.47
	Data Class	76.86	58.45	66.40

Table 3.14: Obtained results by Bayesian approach.

Open source software	Defects	Precision	Recall	$F_{measure}$
GanttProject	Blob	55.54	37.37	44.68
	Functional Decomposition	75.25	82.64	78.77
	Spaghetti Code	82.69	56.25	66.95
	Feature Envy	46.75	34.82	39.91
	Data Class	72.91	51.75	60.53
Log4j	Blob	51.29	35.82	42.18
	Functional Decomposition	70.52	84.45	76.86
	Spaghetti Code	84.99	53.84	65.92
	Feature Envy	69.95	42.59	52.94
	Data Class	66.58	44.81	53.57
Xerces-J	Blob	54.45	33.95	41.82
	Functional Decomposition	83.26	82.59	82.92
	Spaghetti Code	90.27	50.98	65.16
	Feature Envy	57.66	31.25	40.53
	Data Class	72.76	39.75	51.41

Table 3.15: Obtained results by the proposed SP-J48 approach.

Open source software	Defects	Precision	Recall	$F_{measure}$
GanttProject	Blob	70.49	46.48	56.02
	Functional Decomposition	95.90	93.97	94.93
	Spaghetti Code	94.21	92.38	93.29
	Feature Envy	98.90	98.15	98.52
	Data Class	85.07	75.40	79.94
Log4j	Blob	74.42	43.67	55.04
	Functional Decomposition	91.32	89.69	90.50
	Spaghetti Code	93.91	92.75	93.33
	Feature Envy	88.95	89.03	88.99
	Data Class	84.93	73.69	78.91
Xerces-J	Blob	76.69	43.35	55.39
	Functional Decomposition	89.99	88.79	89.39
	Spaghetti Code	92.03	89.88	90.94
	Feature Envy	92.58	95.94	94.23
	Data Class	84.89	72.79	78.38

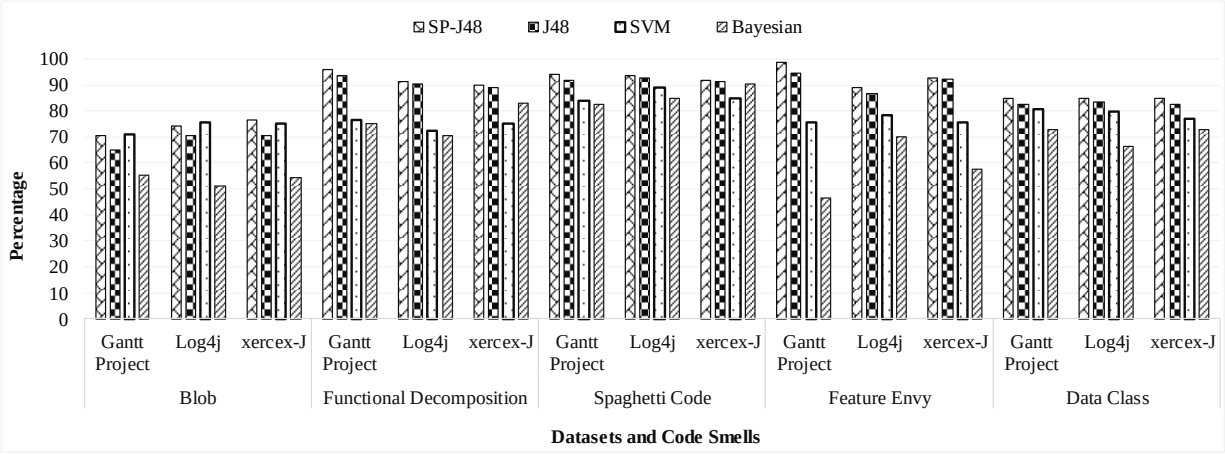


Fig. 3.9. Precision values obtained by proposed and competitor approaches.

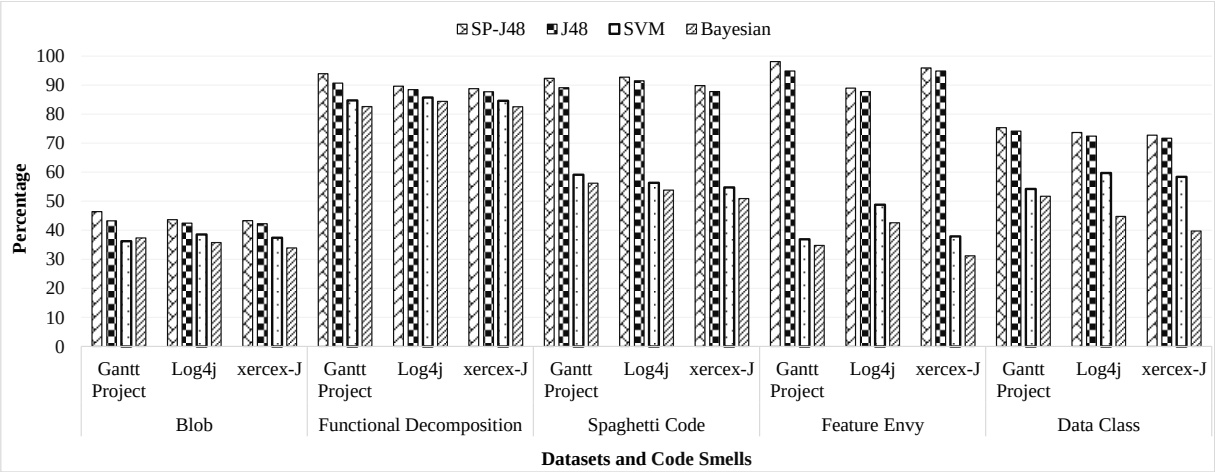


Fig. 3.10. Recall values obtained by proposed and competitor approaches.

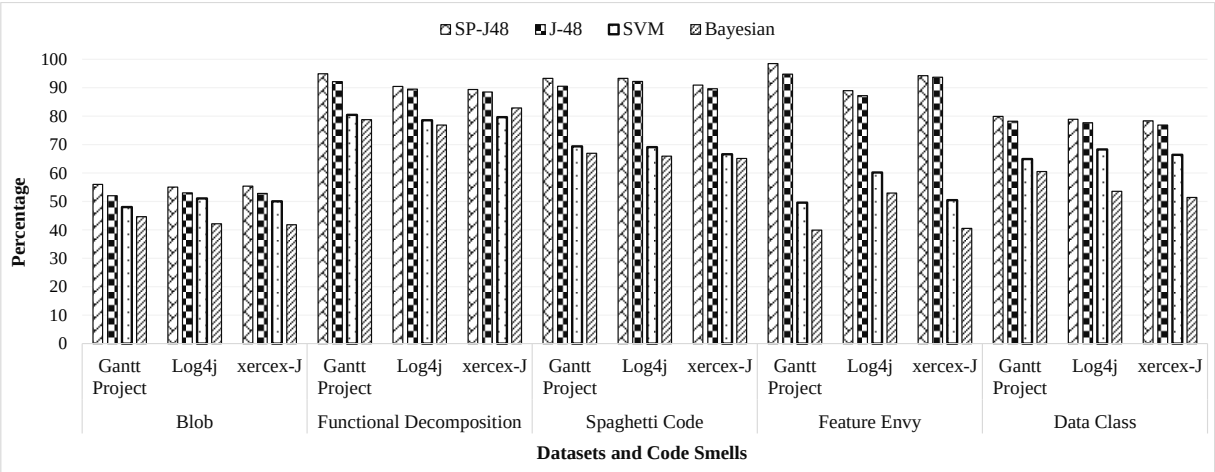


Fig. 3.11. $F_{measure}$ values obtained by proposed and competitor approaches.

3.5 Summary

In this chapter, a novel hybrid algorithm named as SP-J48 is proposed to detect code smells from object-oriented softwares. The proposed algorithm synthesizes the strengths of decision tree based J48 approach and a novel Sandpiper Optimization Algorithm (SPOA). The fundamental concepts of SPOA are inspired from the searching and attacking behaviours of sandpipers. The performance of SPOA is tested on unimodal, multimodal, fixed-dimension multimodal test functions by comparing its obtained outcomes with nine well-established algorithms. In addition, the efficiency of SP-J48 in terms of detecting the code smells is also evaluated on three datasets namely, GanttProject, Log4j, and Xerces-J and the obtained results are compared with three competitor algorithms. The experimental results reveal that the proposed approach is more efficient towards detecting code smells in terms of Precision, Recall, and $F_{measure}$ metrics. Moreover, this hybrid algorithm successfully assisted in removing the parameter tuning problem of existing machine-learning algorithms.

Chapter4

Code Smell Detection using C5.0-SPOA

"The essence of mathematics is not to make simple things complicated, but to make complicated things simple" By S. Gudden

4.1 Overview

In previous Chapter, J48 algorithm is used in conjunction with SPOA called SP-J48, for code smell detection. It has been observed that this algorithm outperforms most of the existing techniques. Despite the fact that J48 is one of the well known algorithm, it suffers from the following limitations:

- **Empty Branches:** In J48, one of the important step is to generate a tree with significant values. In the previous Chapter, there were many nodes with zero and close to zero values but these values do not create or even contribute to create any class in classification. Rather, it makes tree large and complex [171].
- **Insignificant Branches:** In order to build decision tree, the different selected attributes generate the similar number of possible division. But the case is that not all of them are significant for classification. These least significant branches not only lead to decrease in usability of decision tree but also increase the problem of over fitting [172].

Kaur, A., Jain, S. and Goel, S., "**Sandpiper optimization algorithm: a novel approach for solving real-life engineering problems**", *Applied Intelligence*, Springer, (2020), 502, PP. 582-619. **(SCI, Impact Factor: 3.32)**

- **Information Entropy:** There are several problems related to information gain in J48 such as:
  - It do not provide better results when enormous distinct values are used in both discrete and continuous attributes.
  - There is no particular assessment method available to measure the actual information gain before it is applied. The information gain ratio is calculated only after the generation of attribute values. Thus, this disparity or wrong choice of attributes results in less accuracy and performance.
  - It becomes failure, when the information gain is less than the number of attributes used.
  - If the previously selected attributes have less value then it becomes difficult to choose next attributes which leads to decisive selection of attributes thus induce uncertainty.
  - It makes split-up task complex when same valued attributes are used in construction of decision tree; thus, generates unbalanced tree.

In order to address these limitations, C5.0 algorithm in conjunction with SPOA is presented to efficiently identify code smells in software projects.

4.2 Code Smell Detection using C5.0

In literature, many code smell detection approaches have been proposed by the researchers [26, 77, 61], but majority of these techniques are either rule based or metric based. These rules are logical expressions with predefined threshold values. Moreover, the selection of threshold value and software metrics depends upon the experience of development team and is a long trial and error process. Therefore, the efficiency of these approaches depends upon how well the rules are proposed.

In this Chapter to detect the code smells from software systems, the use of C5.0 decision tree algorithm is evaluated. This algorithm automatically selects the metrics and generate rules for code smell detection. A list of eight code smells such as Blob, Large class, Feature Envy, LongParameterList, Data Class, Functional decomposition, Spaghetti Code, and Long method are detected from five open source Java software projects namely, GanttProject, Log4j, Xerces-J (2.7.0), ArgoUML, and Eclipse. The characteristics and detailed description of these projects are mentioned in Tables 4.1 to 4.2.

Table 4.1: Characteristics of five open source softwares.

Name	Version	Lines of Code	Number of Classes
GanttProject	1.10.2	21,267	188
Log4j	1.2.1	10,224	189
Xerces-J	2.7.0	71,217	513
Eclipse	3.8.1	3.5M	10,000
ArgoUML	0.26	300K	2,000

Table 4.2: Description of open source projects.

Projects	Brief Description
GanttProject	GanttProject is an open-source desktop project scheduling and management tool. It is written in Java/Swing. GanttProject is free for any purposes, including commercial use.
Log4j	Log4j is an open source logging framework with fully configurable external configuration files.It allows the developer to control log statements based on application requirement.
Xerces	Xerces is Apache's collection of software libraries for parsing, validating, serializing and manipulating XML. The library implements a number of standard APIs for XML parsing, including DOM, SAX and SAX2. The implementation is available in the Java, C++ and Perl programming languages.
ArgoUML	ArgoUML is an UML diagramming application written in Java and released under the open source Eclipse Public License. By virtue of being a Java application, it is available on any platform supported by Java SE.
Eclipse	Eclipse is an integrated development environment used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment.

In addition, to improve the machine-learning algorithms feature selection can be used to reduce the training time and enhance generalization ability. In order to detect code smells from open source projects an experiment is conducted in

which the performance of decision tree classification is analysed and compared with existing techniques. The following set of steps are followed to perform an experiment.

Step 1: To specify the dataset to be used during performance assessment. It includes selecting the software projects, calculating the software metrics from each class, and identify existing code smells.

Step 2: To execute C5.0 decision tree algorithm.

Step 3: To evaluate the performance of proposed technique using performance measurements such as Precision, Recall, and $F_{measure}$.

The information regarding the existence or not of code smell along with set of several metrics calculated for each class of considered software is provided as an input to C5.0 classification algorithm for code smell detection. The decision tree generated using this information is shown in Fig. 4.1. It is evident from Fig. 4.1

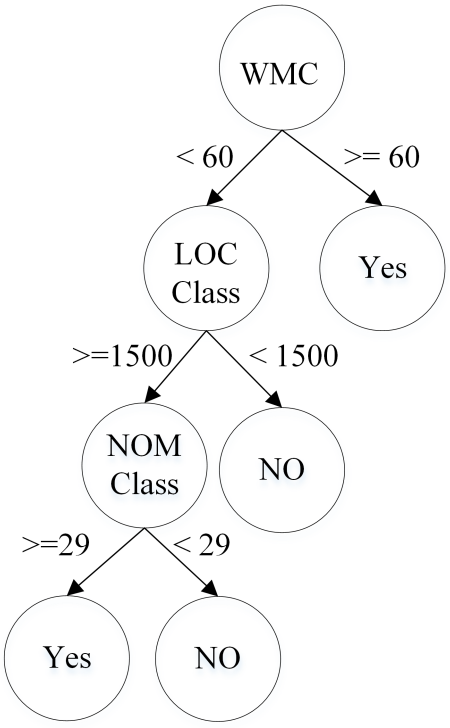


Fig. 4.1. Derived decision tree for Large Class Code smell.

that the presence of code smell in a class can be easily identified from a decision tree. The nodes of a tree are the metrics of subject system. The most significant metrics for detecting code smells are placed at the top of tree by C5.0 and less

significant metrics will appear at lower level and become more specialized as the tree becomes deeper. It is also evident from this experiment that from the set of provided metrics C5.0 not always select the most important metrics. As it is possible to re-adjust the values of given set of metrics and deleting some of them to obtain smaller and accurate tree.

Therefore, to solve the problem of finding the most significant set of metrics a hybrid approach is proposed in this Chapter. This proposed hybrid approach utilizes C5.0 in conjunction with SPOA to automatically achieve the desire task.

Algorithm 5 Hybrid algorithm.

Input: Parameters of C5.0 algorithm, Maximum number of iterations Max_{iter}
Output: Detected code smells
1: **procedure**
2: Choose the initial parameters and normalize it
3: $\alpha \leftarrow$ Initial selection of parameters
4: $iteration \leftarrow 0$
5: **while** $(iteration < Max_{iter})$ **do**
6: $\beta \leftarrow$ Select the best obtained parameters using Algorithm 1
7: $iteration \leftarrow iteration + 1$
8: **end while**
9: $\gamma \leftarrow$ Set of detected code smells (β)
10: return γ
11: **end procedure**

4.3 Empirical Evaluation

4.3.1 Considered Metrics

To obtain the metrics of the classes, two metric extraction tools are used namely, CKJM [166] and POM [167]. Both of these tools are capable of calculating metrics by analyzing the projects .jar files. The analyzed .jar files are obtained from the download pages of each project. The CKJM tool is used to calculate 18 metrics[1] and the POM tool is used to calculate another 44 metrics[2] for each class contained

[1]WMC, DIT, NOC, CBO, RFC, LCOM, Ca, Ce, NPM, LCOM3, LOC, DAM, MOA, MFA, CAM, IC, CBM and AMC

[2]ACAIC, ACMIC, AID, CLD, DCAEC, DCC, DCMEC, DIT, DSC, ICHClass, IR, LCOM1, LCOM2, LCOM5, LOC, MLOCsum, McCabe, NAD NADExtended, NCM, NMA, NMD, NMDExtended, NMI,

in the .jar files, resulting in 62 software metrics. Although some metrics appear repeatedly in the final set, since they are calculated by the two tools. It is necessary to keep the two versions as they are calculated differently by each tool.

Finally, to create a single dataset suitable for use in these experiments, the Bad Smells dataset used in [167] has been joined to the corresponding sets containing the 18 metrics calculated by the CKJM tool, and the 44 metrics calculated by the POM tool, using the class name as the search key for the union. After that, the resulting datasets (one for each software project) are concatenated and the repeated lines are removed leaving only one line for each class name. Some classes found in the Bad Smells dataset are not found in the output of CKJM or POM and vice versa and therefore needed to be removed. Therefore, only 7633 classes present in both datasets are considered for this study.

4.3.2 Research Questions

- **RQ1:** Is C5.0 classification (decision tree) algorithm adequate to recognize code smells?
- **RQ2:** Is It is possible to improve the efficiency of the classification pre-selecting the metrics with SPOA?
- **RQ3:** Does the rules of detection of Bad Smells learned in one project of software preserve their quality when applied to other projects?
- **RQ4:** How do C5.0 algorithm perform when compared with other machine-learning techniques?

To answer the question of RQ1 , the most widely used Kappa measure of agreement is used between the classification models. It verify the compliance of detection obtained by the rules generated by C5.0.

To answer the question of RQ2 , the results of the detection of bad smells are

NMO, NOA, NOC, NOD, NOF, NOH, NOM, NOP, NOPM, NOParam, NOTC, NOTI, RFC_New, SIX , VGsum, WMC1, WMC_New, cohesionAttributes and connectivity

compared using rules generated by the C5.0 receiving as input all the metrics for each class, with the results of detection when the C5.0 receives only the metric pre-selected by SPOA algorithm after different simulation runs.

To answer the question of RQ3 , the set of data is separated for projects of software to perform one experiment where the set of training contains only classes of one particular design, and the sets of tests containing classes another project. The results can be analyzed individually by the value of obtained Kappa measure.

To answer the RQ4, the hybrid approach is compared with existing competitor techniques using four well-known performance metrics namely, Precision, Recall, and $F_{measure}$.

4.3.3 Results and Discussions

4.3.3.1 Experiment 1:

The first experiment used only the C5.0 machine-learning algorithm without the proposed SPOA algorithm to pre-select the metrics. In this experiment, the C5.0 algorithm is configured to perform one cross validation with 10-folds. Table 4.3 shows the results of Experiment 1, performed by C5.0 to generate the rules (Trees of Decision) taking as input the values of all the metrics. In this table, it is seen that, for each one of bad smells, the amount of True negatives, False positives, False negatives, and True Positive, and in the last three columns, the values of Precision, Recall, and Kappa.

4.3.3.2 Experiment 2:

The second experiment is carried out using the proposed SPOA algorithm for pre-selecting the metrics, as shown in Fig. 4.2. It is shown in Figure that firstly algorithm selects the metrics. After that, the proposed SPOA algorithm selects the efficient parameters of C5.0 algorithm. Based on these parameters, C5.0 efficiently detects the code smells. Table 4.4 shows the results of Experiment 2, carried out

Table 4.3: Results of Experiment 1.

Bad smells	TN	FP	FN	TP	Precision	Recall	Kappa
Feature Envy	6875	90	261	407	81.89	60.93	0.67
Blob	6532	146	581	374	71.92	39.16	0.46
Data Class	6600	103	343	587	85.07	63.12	0.69
Functional Decomposition	6782	48	74	729	93.82	90.78	0.91
Large Class	7327	30	57	219	87.95	79.35	0.83
Spaghetti Code	5618	93	190	1732	94.90	90.11	0.90
Long Method	3992	653	626	2362	78.34	79.05	0.65
LongParameterList	6053	22	60	1498	98.55	96.15	0.97

using algorithm C5.0 to generate the rules (Trees of Decision) taking as input only the values of metric pre-selected by SPOA algorithm.

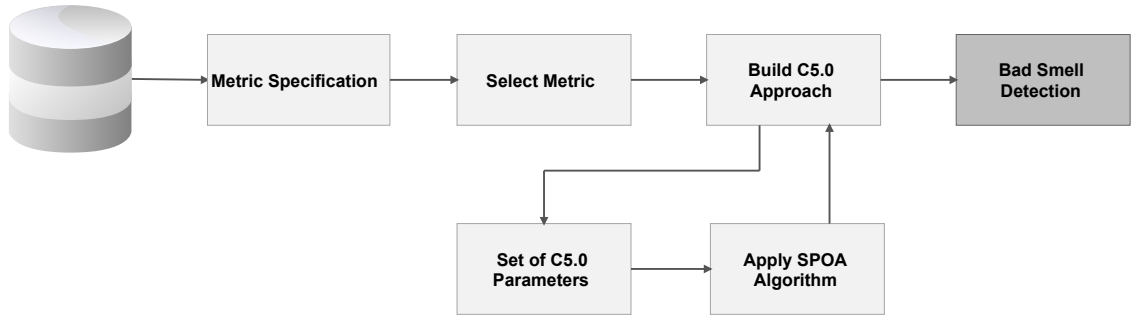


Fig. 4.2. Code smell detection using proposed hybrid approach.

4.3.3.3 Experiment 3:

In the third experiment, with the aim of answering the question of RQ3, one dataset classes is compared with another dataset classes. This experiment is repeated in order to consider all the 20 configurations possible for pairs of sets, such as described in Table 4.5.

For each one of these pairs, the C5.0 has been run 10 times, one for each bad smell. The obtained results in terms of Precision, Recall, and Kappa are described in Tables 4.6 to 4.10.

Table 4.4: Results of Experiment 2.

Bad smells	TN	FP	FN	TP	Precision	Recall	Kappa
Feature Envy	6899	66	247	421	86.45	63.02	0.71
Blob	6583	95	571	384	80.17	40.21	0.49
Data Class	6640	63	320	610	90.64	65.59	0.73
Functional Decomposition	6784	46	62	741	94.16	92.28	0.92
Large Class	7337	20	26	250	92.59	90.58	0.91
Spaghetti Code	5666	45	137	1785	97.54	92.87	0.94
Long Method	4064	581	501	2487	81.06	83.23	0.70
LongParameterList	6063	12	36	1522	99.22	97.69	0.98

Table 4.5: Configuration of pairs of sets of training and testing for the completion of the Experiment 3.

Config. Num.	Set of training	Set of testing
1	GanttProject	Log4j
2	GanttProject	Xerces-J
3	GanttProject	ArgoUML
4	GanttProject	Eclipse
5	Log4j	GanttProject
6	Log4j	Xerces-J
7	Log4j	ArgoUML
8	Log4j	Eclipse
9	Xerces-J	GanttProject
10	Xerces-J	Log4j
11	Xerces-J	ArgoUML
12	Xerces-J	Eclipse
13	ArgoUML	GanttProject
14	ArgoUML	Log4j
15	ArgoUML	Xerces-J
16	ArgoUML	Eclipse
17	Eclipse	GanttProject
18	Eclipse	Log4j
19	Eclipse	Xerces-J
20	Eclipse	ArgoUML

Table 4.6: Results of Experiment 3 using the classes of GanttProject as training set.

Set of test	Bad smells	TN	FP	FN	TP	Precision	Recall	Kappa
Eclipse	Feature Envy	4198	61	391	125	67.20	24.22	0.32
Eclipse	Blob	3947	122	514	192	61.15	27.20	0.31
Eclipse	Data Class	3985	119	392	279	70.10	41.58	0.47
Eclipse	Functional Decomposition	4166	11	264	334	96.81	55.85	0.68
Eclipse	Large Class	4360	409	1	5	1.21	83.33	0.02
Eclipse	Spaghetti Code	2979	0	1755	41	100.00	2.28	0.03
Eclipse	Long Method	2377	111	1174	1113	90.93	48.67	0.45
Eclipse	LongParameterList	3631	2	735	407	99.51	35.64	0.46
ArgoUml	Feature Envy	1909	73	1	4	5.19	80.00	0.09
ArgoUml	Blob	1882	5	64	36	87.80	36.00	0.50
ArgoUml	Data Class	1757	177	27	26	12.81	49.06	0.17
ArgoUml	Functional Decomposition	1885	10	53	39	79.59	42.39	0.54
ArgoUml	Large Class	1809	34	79	65	65.66	45.14	0.51
ArgoUml	Spaghetti Code	1917	3	2	65	95.59	97.01	0.96
ArgoUml	Long Method	1373	150	74	390	72.22	84.05	0.70
ArgoUml	LongParameterList	1670	23	229	65	73.86	22.11	0.29
Log4j	Feature Envy	137	0	39	0	–	0.00	0.00
Log4j	Blob	95	3	63	15	83.33	19.23	0.18
Log4j	Data Class	120	7	29	20	74.07	40.82	0.41
Log4j	Functional Decomposition	113	7	38	18	72.00	32.14	0.31
Log4j	Large Class	115	2	51	8	80.00	13.56	0.15
Log4j	Spaghetti Code	142	2	19	13	86.67	40.63	0.49
Log4j	Long Method	83	52	0	41	44.09	100.00	0.43
Log4j	LongParameterList	124	0	52	0	–	0.00	0.00
Xerces-J	Feature Envy	388	0	107	0	–	0.00	0.00
Xerces-J	Blob	409	15	60	11	42.31	15.49	0.16
Xerces-J	Data Class	353	2	139	1	33.33	0.71	0.00
Xerces-J	Functional Decomposition	438	14	34	9	39.13	20.93	0.23
Xerces-J	Large Class	432	15	42	6	28.57	12.50	0.12
Xerces-J	Spaghetti Code	477	0	6	12	100.00	66.67	0.79
Xerces-J	Long Method	315	19	106	55	74.32	34.16	0.33
Xerces-J	LongParameterList	385	48	49	13	21.31	20.97	0.10

Table 4.7: Results of Experiment 3 using the classes of Log4j as training set.

Set of test	Bad smells	TN	FP	FN	TP	Precision	Recall	Kappa
Eclipse	Feature Envy	4259	0	516	0	–	0.00	0.00
Eclipse	Blob	3922	147	623	83	36.09	11.76	0.11
Eclipse	Data Class	4088	16	603	68	80.95	10.13	0.15
Eclipse	Functional Decomposition	4127	50	417	181	78.35	30.27	0.39
Eclipse	Large Class	4515	254	4	2	0.78	33.33	0.01
Eclipse	Spaghetti Code	2979	0	1762	34	100.00	1.89	0.02
Eclipse	Long Method	2476	12	1889	398	97.07	17.40	0.18
Eclipse	LongParameterList	3380	253	813	329	56.53	28.81	0.26
ArgoUml	Feature Envy	1984	0	3	0	–	0.00	0.00
ArgoUml	Blob	1887	0	93	7	100.00	7.00	0.13
ArgoUml	Data Class	1906	28	37	16	36.36	30.19	0.31
ArgoUml	Functional Decomposition	1891	4	79	13	76.47	14.13	0.23
ArgoUml	Large Class	1823	20	126	18	47.37	12.50	0.17
ArgoUml	Spaghetti Code	1500	420	38	29	6.46	43.28	0.06
ArgoUml	Long Method	1463	60	318	146	70.87	31.47	0.34
ArgoUml	LongParameterList	1687	6	276	18	75.00	6.12	0.09
GanttProject	Feature Envy	190	9	1	0	0.00	0.00	-0.01
GanttProject	Blob	164	36	0	0	0.00	–	0.00
GanttProject	Data Class	139	44	3	14	24.14	82.35	0.28
GanttProject	Functional Decomposition	128	58	0	14	19.44	100.00	0.24
GanttProject	Large Class	181	0	19	0	–	0.00	0.00
GanttProject	Spaghetti Code	139	52	0	2	3.70	100.00	0.05
GanttProject	Long Method	86	79	3	32	28.83	91.43	0.23
GanttProject	LongParameterList	72	120	0	8	6.25	100.00	0.05
Xerces-J	Feature Envy	312	76	76	31	28.97	28.97	0.09
Xerces-J	Blob	334	90	47	24	21.05	33.80	0.10
Xerces-J	Data Class	287	68	90	50	42.37	35.71	0.17
Xerces-J	Functional Decomposition	401	51	2	41	44.57	95.35	0.55
Xerces-J	Large Class	447	0	48	0	–	0.00	0.00
Xerces-J	Spaghetti Code	182	295	0	18	5.75	100.00	0.04
Xerces-J	Long Method	253	81	16	145	64.16	90.06	0.60
Xerces-J	LongParameterList	320	113	9	53	31.93	85.48	0.35

Table 4.8: Results of Experiment 3 using the classes of Xerces-J as training set.

Set of test	Bad smells	TN	FP	FN	TP	Precision	Recall	Kappa
Eclipse	Feature Envy	4259	0	516	0	–	0.00	0.00
Eclipse	Blob	3927	142	623	83	36.89	11.76	0.12
Eclipse	Data Class	4088	16	603	68	80.95	10.13	0.15
Eclipse	Functional Decomposition	4129	48	417	181	79.04	30.27	0.40
Eclipse	Large Class	4515	254	4	2	0.78	33.33	0.01
Eclipse	Spaghetti Code	2979	0	1762	34	100.00	1.89	0.02
Eclipse	Long Method	2476	12	1889	398	97.07	17.40	0.18
Eclipse	LongParameterList	3402	231	813	329	58.75	28.81	0.27
ArgoUml	Feature Envy	1905	77	1	4	4.94	80.00	0.09
ArgoUml	Blob	1792	95	67	33	25.78	33.00	0.25
ArgoUml	Data Class	1768	166	18	35	17.41	66.04	0.24
ArgoUml	Functional Decomposition	1839	56	24	68	54.84	73.91	0.61
ArgoUml	Large Class	1841	2	143	1	33.33	0.69	0.01
ArgoUml	Spaghetti Code	630	1290	0	67	4.94	100.00	0.03
ArgoUml	Long Method	620	903	2	462	33.85	99.57	0.24
ArgoUml	LongParameterList	1602	91	226	68	42.77	23.13	0.22
GanttProject	Feature Envy	199	0	1	0	–	0.00	0.00
GanttProject	Blob	200	0	0	0	–	–	–
GanttProject	Data Class	177	6	16	1	14.29	5.88	0.04
GanttProject	Functional Decomposition	186	0	14	0	–	0.00	0.00
GanttProject	Large Class	181	0	19	0	–	0.00	0.00
GanttProject	Spaghetti Code	191	0	9	0	–	0.00	0.00
GanttProject	Long Method	165	0	35	0	–	0.00	0.00
GanttProject	LongParameterList	192	0	8	0	–	0.00	0.00
Log4j	Feature Envy	136	1	28	11	91.67	28.21	0.37
Log4j	Blob	98	0	42	36	100.00	46.15	0.49
Log4j	Data Class	120	7	27	22	75.86	44.90	0.45
Log4j	Functional Decomposition	114	6	29	27	81.82	48.21	0.49
Log4j	Large Class	117	0	31	28	100.00	47.46	0.55
Log4j	Spaghetti Code	135	9	17	15	62.50	46.88	0.45
Log4j	Long Method	124	11	2	39	78.00	95.12	0.81
Log4j	LongParameterList	122	2	24	28	93.33	53.85	0.60

Table 4.9: Results of Experiment 3 using the classes of ArgoUML as training set.

Set of test	Bad smells	TN	FP	FN	TP	Precision	Recall	Kappa
Eclipse	Feature Envy	4259	0	516	0	–	0.00	0.00
Eclipse	Blob	4068	1	701	5	83.33	0.71	0.01
Eclipse	Data Class	4043	61	554	117	65.73	17.44	0.23
Eclipse	Functional Decomposition	4176	1	537	61	98.39	10.20	0.17
Eclipse	Large Class	4643	126	4	2	1.56	33.33	0.03
Eclipse	Spaghetti Code	2964	15	1516	280	94.92	15.59	0.18
Eclipse	Long Method	2406	82	1780	507	86.08	22.17	0.19
Eclipse	LongParameterList	3628	5	999	143	96.62	12.52	0.18
GanttProject	Feature Envy	188	11	1	0	0.00	0.00	-0.01
GanttProject	Blob	164	36	0	0	0.00	–	0.00
GanttProject	Data Class	161	22	7	10	31.25	58.82	0.33
GanttProject	Functional Decomposition	159	27	0	14	34.15	100.00	0.45
GanttProject	Large Class	153	28	0	19	40.43	100.00	0.51
GanttProject	Spaghetti Code	191	0	9	0	–	0.00	0.00
GanttProject	Long Method	126	39	11	24	38.10	68.57	0.34
GanttProject	LongParameterList	137	55	0	8	12.70	100.00	0.17
Log4j	Feature Envy	137	0	39	0	–	0.00	0.00
Log4j	Blob	98	0	78	0	–	0.00	0.00
Log4j	Data Class	122	5	39	10	66.67	20.41	0.21
Log4j	Functional Decomposition	119	1	42	14	93.33	25.00	0.30
Log4j	Large Class	111	6	40	19	76.00	32.20	0.32
Log4j	Spaghetti Code	142	2	20	12	85.71	37.50	0.46
Log4j	Long Method	100	35	24	17	32.69	41.46	0.14
Log4j	LongParameterList	124	0	52	0	–	0.00	0.00
Xerces-J	Feature Envy	388	0	107	0	–	0.00	0.00
Xerces-J	Blob	424	0	71	0	–	0.00	0.00
Xerces-J	Data Class	340	15	138	2	11.76	1.43	-0.04
Xerces-J	Functional Decomposition	451	1	40	3	75.00	6.98	0.11
Xerces-J	Large Class	442	5	25	23	82.14	47.92	0.57
Xerces-J	Spaghetti Code	379	98	16	2	2.00	11.11	-0.03
Xerces-J	Long Method	327	7	149	12	63.16	7.45	0.07
Xerces-J	LongParameterList	433	0	58	4	100.00	6.45	0.11

Table 4.10: Results of Experiment 3 using the classes of Eclipse as training set.

Set of test	Bad smells	TN	FP	FN	TP	Precision	Recall	Kappa
ArgoUml	Feature Envy	1895	89	1	2	2.20	66.67	0.04
ArgoUml	Blob	1770	117	67	33	22.00	33.00	0.22
ArgoUml	Data Class	1768	166	18	35	17.41	66.04	0.24
ArgoUml	Functional Decomposition	1831	64	24	68	51.52	73.91	0.58
ArgoUml	Large Class	1841	2	143	1	33.33	0.69	0.01
ArgoUml	Spaghetti Code	630	1290	0	67	4.94	100.00	0.03
ArgoUml	Long Method	420	1103	2	462	29.52	99.57	0.15
ArgoUml	LongParameterList	1590	103	226	68	39.77	23.13	0.21
GanttProject	Feature Envy	186	13	1	0	0.00	0.00	-0.01
GanttProject	Blob	164	36	0	0	0.00	–	0.00
GanttProject	Data Class	176	7	5	12	63.16	70.59	0.63
GanttProject	Functional Decomposition	180	6	2	12	66.67	85.71	0.73
GanttProject	Large Class	125	56	0	19	25.33	100.00	0.30
GanttProject	Spaghetti Code	191	0	9	0	–	0.00	0.00
GanttProject	Long Method	157	8	3	32	80.00	91.43	0.82
GanttProject	LongParameterList	182	10	0	8	44.44	100.00	0.59
Log4j	Feature Envy	135	2	35	4	66.67	10.26	0.13
Log4j	Blob	98	0	78	0	–	0.00	0.00
Log4j	Data Class	126	1	37	12	92.31	24.49	0.31
Log4j	Functional Decomposition	119	1	40	16	94.12	28.57	0.34
Log4j	Large Class	83	34	40	19	35.85	32.20	0.03
Log4j	Spaghetti Code	144	0	32	0	–	0.00	0.00
Log4j	Long Method	108	27	13	28	50.91	68.29	0.43
Log4j	LongParameterList	124	0	48	4	100.00	7.69	0.11
Xerces-J	Feature Envy	388	0	107	0	–	0.00	0.00
Xerces-J	Blob	408	16	63	8	33.33	11.27	0.10
Xerces-J	Data Class	352	3	139	1	25.00	0.71	0.00
Xerces-J	Functional Decomposition	443	9	36	7	43.75	16.28	0.20
Xerces-J	Large Class	433	14	36	12	46.15	25.00	0.27
Xerces-J	Spaghetti Code	477	0	6	12	100.00	66.67	0.79
Xerces-J	Long Method	332	2	141	20	90.91	12.42	0.15
Xerces-J	LongParameterList	377	56	45	17	23.29	27.42	0.13

4.3.3.4 Experiment 4:

The comparison of the C5.0-SPOA (proposed technique) with the existing approaches in terms of Precision, Recall, and $F_{measure}$ is shown in Tables 4.12 to 4.15. The values exposes the accuracy and efficiency of C5.0-SPOA over other well-known methods.

4.3.3.5 Evaluation of Results

It is observed from the Tables 4.12 to 4.15 and Figs. 4.3 to 4.5 that in case of Precision metric, the proposed C5.0-SPOA algorithm generates better values for all eight code smells. The Precision value of proposed C5.0-SPOA for Feature envy, Blob, Data Class, Functional decomposition, Large Class, Spaghetti Code, Long Method, and LongParameterList is 86%, 80%, 90%, 94%, 92%, 97%, 81%, and 99%, respectively. Following this, the second best values for Precision metric are bagged by C5.0 algorithm. After this, J48 and SVM provides better results followed by Bayesian algorithm.

Whereas, Recall values generated by C5.0-SPOA algorithm are better for seven out of eight code smells. It generates 63%, 40%, 66%, 92%, 91%, 93%, 83%, and 98% Recall for Feature envy, Blob, Data Class, Functional decomposition, Large Class, Spaghetti Code, Long Method, and LongParameterList, respectively. In addition, C5.0 secured second position in providing the better results followed by J48, SVM, and Bayesian, respectively.

Moreover, the $F_{measure}$ values of C5.0-SPOA are also superior than other competitor algorithms. $F_{measure}$ values for C5.0-SPOA are 72%, 53%, 76%, 93%, 91%, 95%, 82%, and 98% for Feature envy, Blob, Data Class, Functional decomposition, Large Class, Spaghetti Code, Long Method, and LongParameterList, respectively. Whereas, for C5.0, $F_{measure}$ values are 69%, 50%, 72%, 92%, 83%, 92%, 78%, and 97%, respectively, which are better than J48, SVM and Bayesian algorithms.

Table 4.11: The obtained results by C5.0 approach.

Bad smells	Precision	Recall	$F_{measure}$
Feature Envy	81.89	60.93	69.87
Blob	71.92	39.16	50.71
Data Class	85.07	63.12	72.47
Functional Decomposition	93.82	90.78	92.28
Large Class	87.95	79.35	83.43
Spaghetti Code	94.90	90.11	92.45
Long Method	78.34	79.05	78.69
LongParameterList	98.55	96.15	97.34

Table 4.12: The obtained results by J48 approach.

Bad smells	Precision	Recall	$F_{measure}$
Feature Envy	82.90	62.91	69.99
Blob	74.24	40.13	52.98
Data Class	87.09	65.11	74.89
Functional Decomposition	94.55	92.11	94.09
Large Class	89.90	80.30	84.40
Spaghetti Code	95.34	92.10	93.24
Long Method	80.11	78.15	80.61
LongParameterList	95.52	92.10	91.30

Table 4.13: The obtained results by SVM approach.

Bad smells	Precision	Recall	$F_{measure}$
Feature Envy	72.56	33.23	45.58
Blob	75.34	34.66	47.48
Data Class	79.47	61.55	69.37
Functional Decomposition	71.92	58.28	64.39
Large Class	88.29	56.72	69.07
Spaghetti Code	85.95	53.37	65.85
Long Method	75.18	59.41	66.37
LongParameterList	86.53	61.84	72.13

Table 4.14: The obtained results by Bayesian approach.

Bad smells	Precision	Recall	$F_{measure}$
Feature Envy	23.42	94.03	37.50
Blob	56.31	32.58	41.28
Data Class	51.26	38.35	43.88
Functional Decomposition	30.54	91.84	45.84
Large Class	66.83	73.67	70.08
Spaghetti Code	49.34	32.71	39.34
Long Method	63.79	44.16	52.19
LongParameterList	48.45	39.24	43.36

Table 4.15: The obtained results by the proposed C5.0-SPOA approach.

Bad smells	Precision	Recall	$F_{measure}$
Feature Envy	86.45	63.02	72.90
Blob	80.17	40.21	53.56
Data Class	90.64	65.59	76.11
Functional Decomposition	94.16	92.28	93.21
Large Class	92.59	90.58	91.58
Spaghetti Code	97.54	92.87	95.15
Long Method	81.06	83.23	82.13
LongParameterList	99.22	97.69	98.45

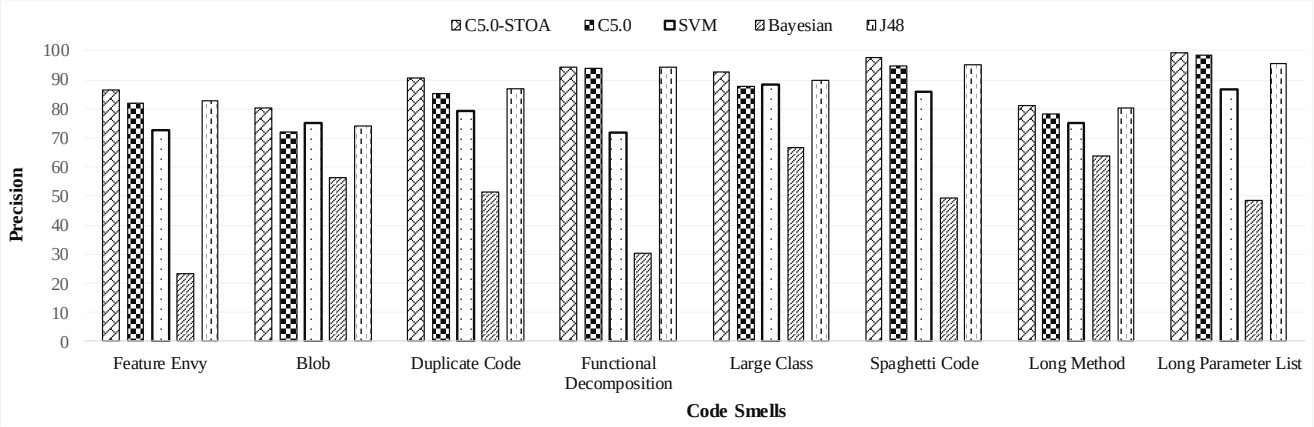


Fig. 4.3. Precision values obtained by proposed and other competitive algorithms.

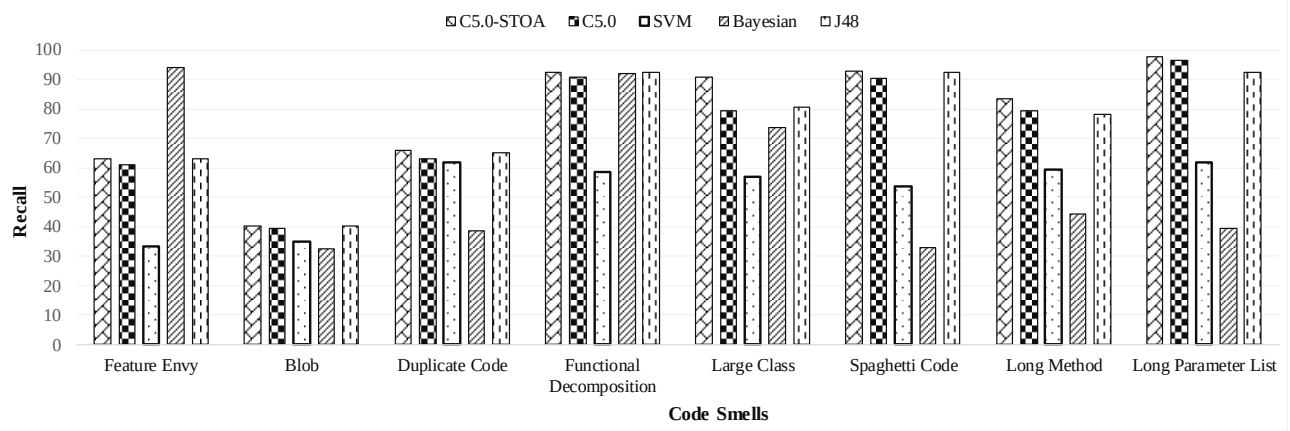


Fig. 4.4. Recall values obtained by proposed and other competitive algorithms.

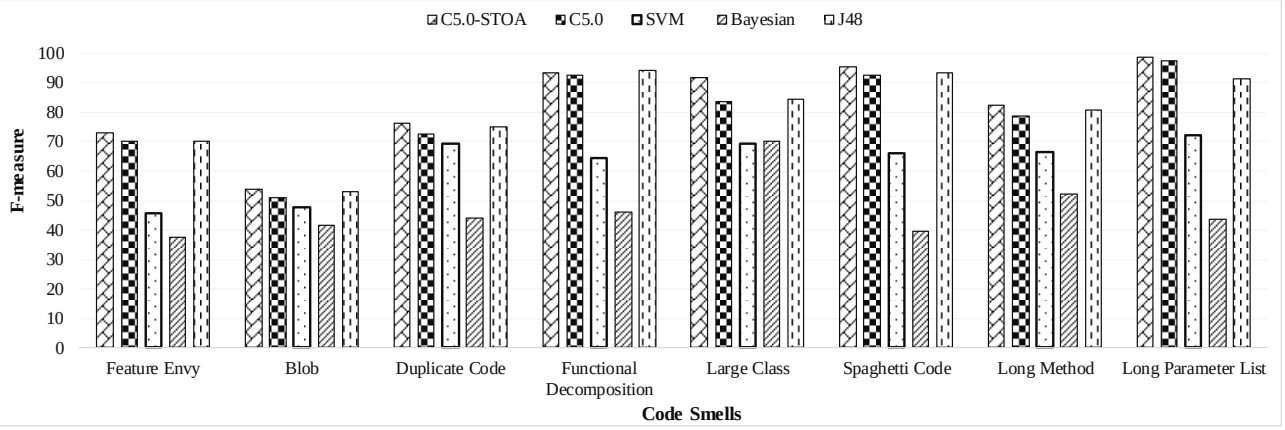


Fig. 4.5. $F_{measure}$ values obtained by proposed and other competitive algorithms.

4.4 Summary

This chapter firstly enlists the motivations behind hybridizing C5.0 with SPOA instead of using SP-J48. Later, the code smell detection process utilizing the new hybrid algorithm named C5.0-SPOA is explained in detail. The empirical evaluation of the projected approach is performed on five datasets employing three performance measures. The experimental outcomes indicate that C5.0-SPOA is able to efficiently identify the code smells and outperforms the other code smell detection techniques.

Chapter5

Impact of Code Smell Prioritization on Software Quality

"It is not the answer that enlightens, but the question." By E. I. Decouvertes

5.1 Overview

Prioritization is the process of determining the order in which the code smells should be removed from the software projects so as to enhance the software quality. Once the code smells are identified, these should be ranked according to their significance. Many authors have reported that refactoring itself is a time-consuming and expensive phenomenon. It is a challenging task for the developer to judge which code smell to refactor first, and what type of refactoring solutions to apply without measurable evidence on the impact of refactoring. The following points constitutes the motivations for this work:

- Although existing research provided tools for code smell detection or the automatic or semi-automatic applications of refactorings for their removal, most of these tools do not prioritize the code smell based on their harmful effect. Therefore, without understanding the harmful effect of a smell, schedule-bound industry practitioners feel reluctant to remove them because

Kaur, A., Jain, S. and Goel, S., "**Sandpiper optimization algorithm: a novel approach for solving real-life engineering problems**", *Applied Intelligence*, Springer, (2020), 502, PP. 582-619. **(SCI, Impact Factor: 3.32)**

of perceived cost, effort, and time intensive outcome of this process.

- Most of the existing techniques utilize manual approach to prioritize the code smells, which are generally considered as time consuming and error intensive.

Considering the aforementioned problems this Chapter presents a technique that automatically prioritize the code smells based on three factors namely, Versioning History (VH), Architectural Relevance (AR), and Code Smell Relevance (CSR). The major motivations behind considering these factors are as follows:

- When only the source code of the software is available, the historical information of the system can be used to predict the code smell that requires immediate refactoring.

- The classes that were frequently refactored in the past are more likely to undergo refactoring in the future [173]. Conversely, if some classes remain unmodified over the previous versions, it would not be a top priority for the developer to perform refactoring for their removal. Hence, such classes can be neglected while identifying top priority code elements for immediate refactoring treatments.

- Each code smell has its own severity score that facilitates developers to immediately correct the most critical code smell [174]. Thus, relying solely on the presence of code smells in a class, when identifying the critical code smells is not considered as a good practice. Code smells should be ranked by determining the severity of each code smell present in the class.

- In software industry, many projects have been terminated or discontinued as a consequence of architectural defects in the software. Architectural defects have a more detrimental effect on the software quality than traditional code defects. Thus, for the longevity of software systems, the architectural problems must be fixed first. Developers need to implement architecturally

relevant strategies for code refactoring.

Considering multiple criterions help us to analyse the code smells from diverse aspects and to explore whether code smell is a serious issue.

5.1.1 Problem Formulation

Refactoring is not freely implemented or adopted in software industries by the software development team, either due to limited resources or hard project deadlines and work pressure. Hence, they look for optimal refactoring activities that would acquire negligible effort while outputting decent benefits in terms of enhanced software quality.

In this work, a technique is proposed that recommends a ranking of code smells based on a combination of three criteria namely, architectural relevance, code smell relevance for the system, and past history of a class in which smell is present. This work looks for a solution in the Law of the Vital Few, which states - "only 20 % of code contains 80% of errors" [34]. This technique is capable of detecting, at the top of the generated code smells priority list, a set of refactoring-prone (based on historical data), and most crucial (based on architectural relevance and code smell information) application classes that need immediate refactoring. The overall process is given in Fig. 5.1.

The first two steps involve considering the relevant classes and eliminating architectural irrelevant classes. Third step is concerned with finding code smell severity and the last step involves calculating the rank of each detected code smell. Furthermore, to evaluate the performance of proposed approach three metrics namely, code smells correction ratio (CSCR), estimated effort(EE), and severity of fixed code smells of a system (SFCS) are considered along with the mechanism followed by Ouni et al. [175, 176, 131]. The process of generating a code smell sequence, i.e., order of code smell prioritization to eradicate code smell, involves the inspection of vast search space which not only include the sequence in which

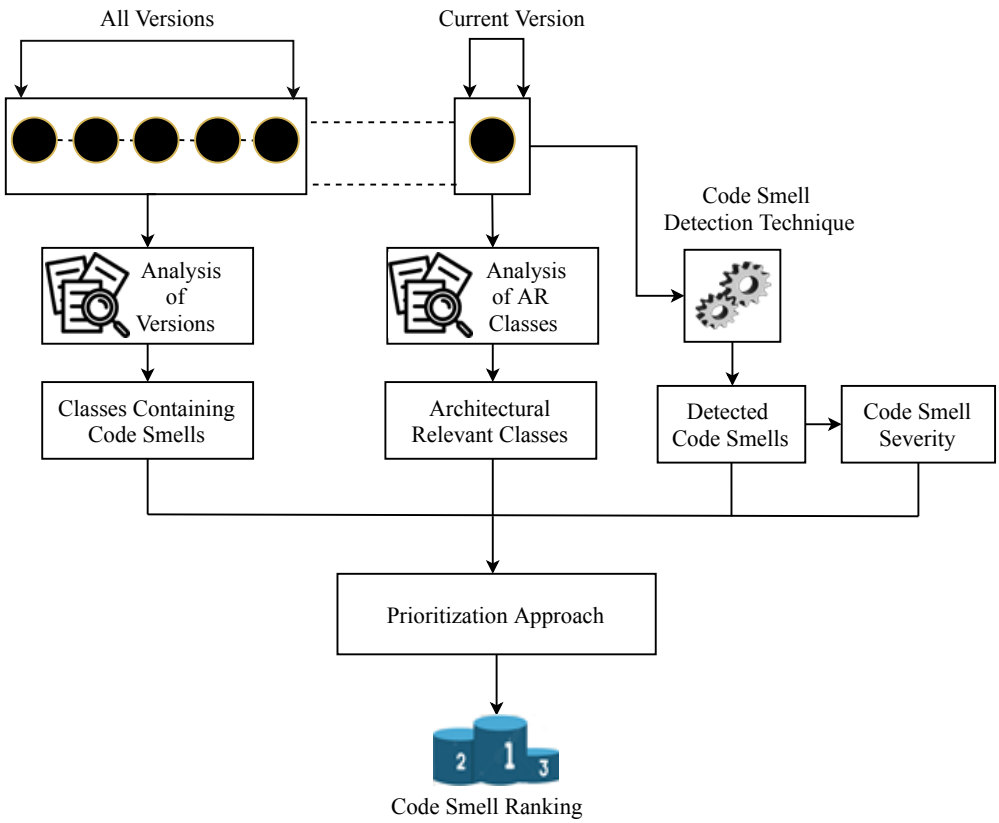


Fig. 5.1. Code smell prioritization process.

these code smells are prioritized but also restricted to code smells combination count. Therefore, optimization algorithms are employed to obtain code smells sequence.

In this work, an optimization algorithm named "SPOA" is used to obtain code smells prioritisation. To implement this, three different parameters are used namely, code smell relevance, versioning history, and architectural relevance. To optimize these aforementioned parameters we have converted them into a fitness function as shown in Eq. (5.1). These parameter are explained in detail in following subsection.

$$CodeSmellScore = x \times (VH \times CSR) + (1 - x) \times AR,\ where\ 0 \leqslant x \leqslant 1 \tag{5.1}$$

The optimization process of SPOA starts by producing a set of random solutions as initial population. Here, solutions refer to sequence of n code smells. Following

this, performance of each solution is assessed using fitness function defined in Eq. (5.1). The best search agent is examined from the given search space. The other search agents update their positions with respect to the obtained best solution. After this mechanism, the fitness value of each search agent is again calculated. Furthermore, if there exists a better solution than previous optimal solution, the algorithm firstly update the best search agent and then update the other search agents accordingly. SPOA algorithm will continue till the stopping criteria is not reached. Finally, upon the satisfaction of termination criteria, the best positions corresponding to search agents are fetched as optimal solution.

5.1.2 Fitness Parameters

The detail of considered fitness parameters is as follows:

- Versioning History: It is assumed that prioritization of code smells for refactoring is more effective when it targets the classes that have changed in considerable volume in its past generations. On the other hand, it would not be the top priority of the developers to remove code smells from those classes which have not changed in the past versions. The main motive behind considering this point is that prioritization is a kind of preventive maintenance [177]. Thus, it is understandable to prioritize code smells which indicate the classes that experienced refactoring in the past. In preventive maintenance, while solving the design problems the uncertainty lies in making decisions regarding resources and effort investment.

  Thus, future maintainability of a software can potentially be improved by prioritization; i.e., prioritization of code smells includes a form of predicting about future changes. Analysing the old versions of a software helps the software developers to find the previously frequently refactored classes. These classes can be classified as change-prone classes [178]. The left-over classes that are not refactored in the previous generations are assumed as less harmful. Therefore, those classes are screened at this stage.

In this chapter, code smell volatility is based on the changes that includes a class influenced by code smells. Suppose there are three successive versions of a software such as S-1, S, S+1, changes in these versions can be defined as ($Difference_{S-1,S}$) which includes change between versions S-1 and S and ($Difference_{S,S+1}$), i.e., change between S version and S+1. Thus, the total change in class A will be the sum of number of times refactoring applied to class A between versions S-1, S+1. The process of calculating change history between two successive versions of a software is shown in Fig. 5.2. The information regarding change in versions of a software can be gathered

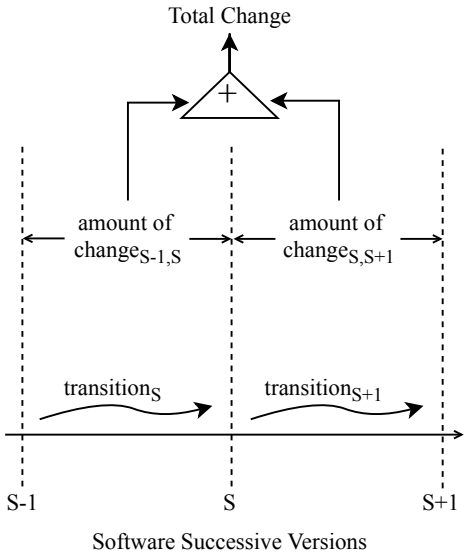


Fig. 5.2. Code smell volatility calculation.

from software repositories such as GitHub, SourceForge, and so on. But sometimes this information is not up to date or is unavailable, therefore, the information regarding the refactoring occurred between two successive software versions can be gathered using automatic tools namely Ref-Finder[1] and RefactoringCrawler[2].

In this work, Ref-Finder tool is used to find the frequently refactored class in two software successive versions. Ref-Finder tool is an Eclipse Plugin[3]

[1]https://sites.google.com/site/reffindertool/
[2]http://dig.cs.illinois.edu/tools/RefactoringCrawler/
[3]https://eclipse.org/

that utilizes template based refactoring (restructuring) and logic-meta programming method to detect complicated refactorings among two versions of a software. It also endorse Fowler's [29] 63 refactoring activities and have 79% overall Precision. Two subsequent software versions are given to the tool as an input which automatically detects the changed classes as an output. It also specifies the type of code smell removed along with the location from where it is removed. Therefore, using this tool, the information regarding changed classes is collected.

Furthermore, the total class volatility is computed using Eq. (5.2). According to historical average model, it is considered that conditional mean is constant. Thus, the best method to calculate total volatility is to given by calculating the average of previous volatilities.

$$VH = \frac{1}{t}\sum_{s=1}^{t} t(A) \tag{5.2}$$

where t indicates the number of previous versions considered for calculating the average, $t(A)$ defines the number of times a class A was changed in the past. For example, we are using lines of code (LOC) metric to analyze the change in a class. Consider there are three versions of a system such as V_1, V_2 and V_3 and code smell is found in class A having LOC values 15, 20 and 25, respectively for the three versions of a system. Then, using this LOC values, the percentage of change in class will be $\frac{20\times100}{15} - 100 = 33.3\%$ for version V_2 and $\frac{25\times100}{20} - 100 = 25\%$ for version V_3. Therefore, total VH will be $\frac{33.3+25}{2} = 29.15\%$ [132]. The details are shown in Table 5.1.

Table 5.1: Example of changed class analysis.

Metrics	v_1	v_2	v_3
LOC	15	20	25
% of change in a class	-	33.3	25

Each time a class is found to be changed while going from one version to

another, its volatility score is incremented by 1. The classes of a current version of a software which are changed even once are stored along with their respective frequency score. While assessing the versions, only those classes are taken into account which have been changed more than once. But it would also include the classes that have experienced refactoring just once in the past versions. Thus, to overcome this shortcoming, the classes are manually examined and assured whether to include or discard those classes.

- Code Smell Relevance (CSR): It defines the extent to which these smells are harmful to system's design. While prioritizing the code smells for refactoring, the severity of code smells is an another significant part to take into consideration as it provides the prioritization of refactoring efforts. Different code smell instances have different severity or intensity and size, and their impact on software quality is also different, thus they should be handled in different means. In a code smell instance, severity is defined as an approximation of total number of these characteristics [140]. For example, if a God Class is very large and complex, it is considered to be have high severity. Finding the code smell severity helps to facilitate reliable information to the project developers, permitting them to prioritize re-engineering and refactoring efforts, i.e., suggesting them to refactor high severity code smells first.

  This second parameter defines how significant a code smell is for the developer. It allows the developer to assign a severity to each code smell. It generally differs from developer to developer, and in fact a developer's estimation of smell severity also get changed with time. In this research, severity score to each smell is assigned between value 0 and 1 based on the recommendations of Mkaouer et al. [179]. The severity scores used for Blob, Feature Envy, Spaghetti Code, Functional Decomposition, and Data Class are 0.8, 0.7, 0.6, 0.4 and 0.3, respectively. The total severity of a class containing code smell can be defined as sum of the severity of each code smell present

in it as shown in Eq. (5.3).

$$TotalSeverity(A) = \sum_{i=1}^{k} SmellSeverity(d_i) \tag{5.3}$$

where k represents the total number of code smells and d_i represents the i^{th} code smell.

- Architectural Relevance (AR): The third major parameter considered for analysing the classes containing code smells for refactoring is architectural relevance. As compared to traditional code smells architectural problems have more dominating impact on lifecycle and quality of software [180]. Thus, only those classes that have direct connection with architectural issues are selected [181]. These classes are analysed from the current version of a software because they believed to be base classes of a software and delay in their refinement/betterment can cause worsening impact on software quality.

  The classes of current version of a software are analysed using Organic tool[4]. Organic tool is an Eclipse plugin that helps to identify architectural relevant code smells from the source code of a software. It classifies the code anomaly agglomerations into different topologies namely, hierarchical, cross-boundary, intra-boundary, and concern based. But we have considered the results based on only two topologies such as intra-boundary and cross boundary topologies, as these topologies provide greater number of architectural relevant problems [181].

The classes identified using VH and AR parameters are collected and then common set of classes (classes which are both architectural relevant and frequently changed) are short listed. Furthermore, these common set of classes along with the given severity of code smells are given as an input to the proposed approach in order to generate the rank of code smells.

[4]http://wnoizumi.github.io/organic/plugin

5.1.3 A Hypothetical Example

To understand the working of proposed approach, a hypothetical example with assumed data values is sketched and explained in this section. Consider a software program V with ten classes and four previous versions namely, V_1,V_2,V_3,V_4, and a current version V_5 whose code smells needs to be prioritized for refactoring. Suppose, in each of the five versions of software V, the constituent classes are as follows:

V_1 - C_1, C_2, C_3, C_4, C_5, C_6

V_2 - C_1, C_2, C_3, C_5, C_6, C_7, C_8

V_3 - C_1, C_2, C_3, C_5, C_6, C_7, C_8

V_4 - C_1, C_3, C_5, C_6, C_7, C_8, C_9, C_{10}, C_{11}

V_5 - C_1, C_3, C_5, C_6, C_7, C_8, C_{11}, C_{12}, C_{14}, C_{15}

For this example, the process of proposed approach at each stage is shown in Fig. 5.3. Here, rectangle ($\square$) represents the actual steps of contemplated approach. Diamond ($\diamond$) represents different versions of the software. Dotted arrows ($\dashrightarrow$) reflect the execution of refactoring activities between two succeeding software version classes. Strong arrows ($\rightarrow$) reflect the changed classes while the software is being versioned.

During software versions analysis, the change history of each class present in version V_5 is examined. A matrix is created that contains all 10 classes of V_5 and evaluates each subsequent version change to calculate the frequency score of each class. As classes C_1, C_3, C_5, C_6, C_7, C_8, C_{11} are found to be changed at least once, so these classes are kept in database and remainder classes are rejected at this stage.

Furthermore, after examining the classes of version V_5, classes C_1, C_3,C_6,C_8, C_{11} are found to be faulty as well as relevant from architectural perspective. Therefore, only these classes should be considered for further analysis. In addition, a common set of classes are selected as an intermediate step based on their change-proneness

and architectural relevance. Lastly, ranks are generated on the basis of three parameters, namely Versioning History, Code Smell Relevance, and Architectural Relevance.

5.2 Empirical Evaluation

The considered code smells, subject systems used, and performance metrics for evaluating the performance of presented approach are discussed in the following subsections. The results of proposed approach are also compared with existing techniques.

5.2.1 Considered Code Smells and Subject Systems

The proposed hybrid approach for code smell detection is used to detect Blob, Functional Decomposition, Spaghetti Code, Feature Envy, and Data Class code smells. Using SPOA, these code smells are further prioritized in order to help the developers to refactor the critical one. The detailed description of these code smells is given in Chapter 1. In addition, the performance of underlying approach for code smell prioritization is evaluated on three Java based open source softwares namely, GanttProject, Xerces-J, and Log4j. The detailed description of these datasets is given in Chapter 3. Table 5.2 provides the general characteristics and versions of the selected systems such as Kilo Lines Of Code (KLOC)in source code, total number of classes present in the software and code smells present in each version. Overall 30 versions of all three systems with multiple KLOC are analysed.

5.2.2 Evaluation Metrics

After every step, four metrics namely, CSCR [175], EE [176], and SFCS [179] are calculated to evaluate the performance of proposed approach. CSCR is defined as the ratio of total number of code smell instances to be refactored by prioritizing the smells, to the total number of code smell instances present in the system. The

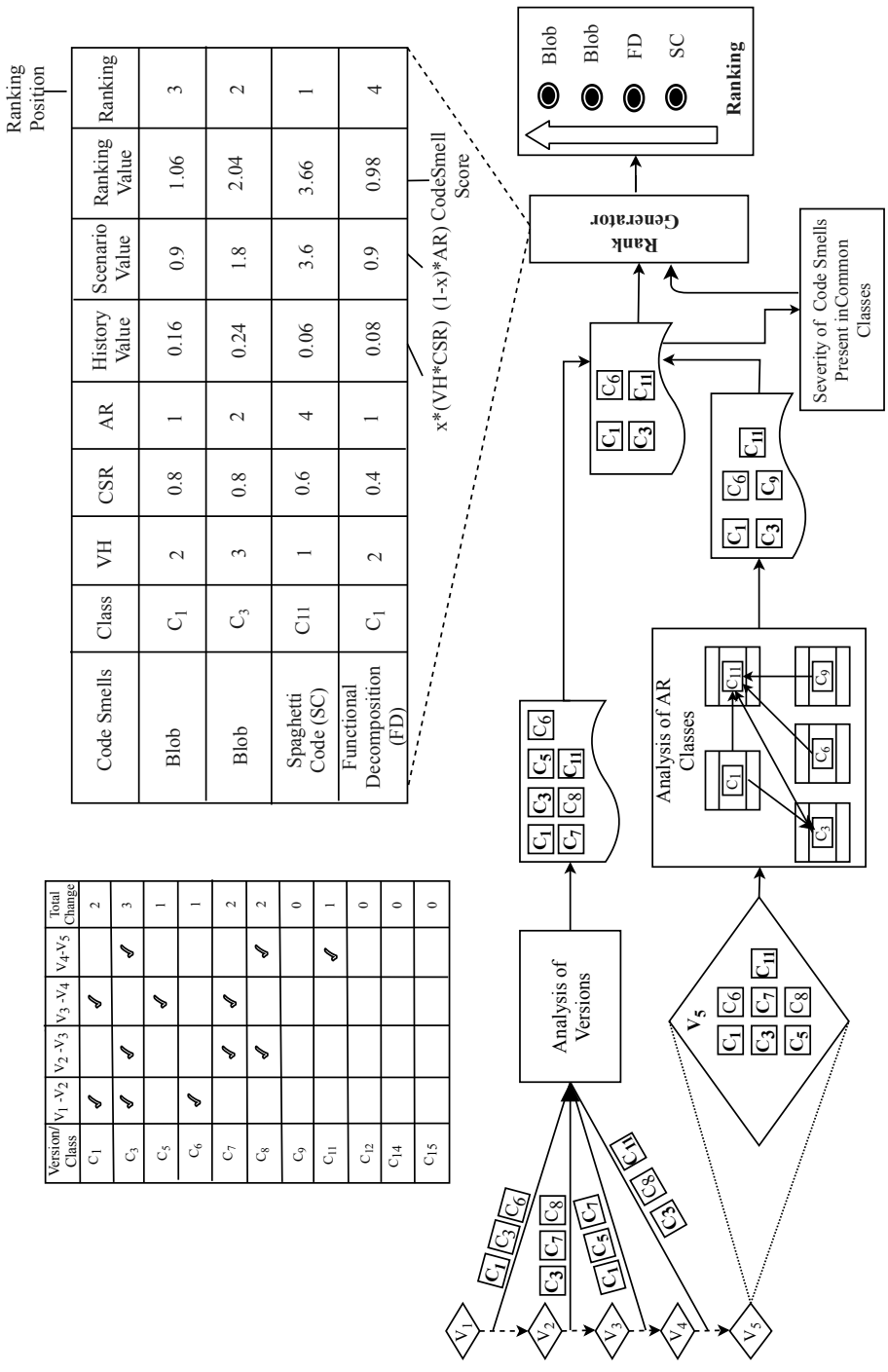


Fig. 5.3. Hypothetical example illustrating the working approach.

Table 5.2: Characteristics of sample applications.

Dataset/Versions	# Classes	#KLOC	#Code Smells
GanttProject			
V2.6.1	498	64	542
V2.6.2	500	65	597
V2.6.3	503	65.5	586
V2.6.4	508	66	602
V2.6.5	512	67	620
V2.6.6	510	67.3	600
V2.7.0	516	68	642
V2.7.1	519	68.4	655
V2.7.2	522	69.4	669
V2.8.0	532	71	671
Log4j			
V1.2.15rc6	189	21	15
V1.2.15	192	23	17
V1.2.16rc1	198	24.5	19
V1.2.16rc2	205	27	22
V1.2.16	215	28	68
V1.2.17	227	32.5	95
V1.2.17rc1	240	34	120
V1.2.17rc2	245	36	165
V1.2.17rc3	270	36.5	176
V1.2.17	296	38	192
Xerces-J			
V1.6	190	62	66
V1.7	210	68	82
V2.0	290	91.5	139
V2.1	280	120	171
V2.2	285	150	192
V2.3	300	175	280
V2.4	340	184	241
V2.5	380	195	250
V2.6	420	201	276
V2.7	513	240	325

formula to calculate CSCR is given in Eq. (5.4).

$$CSCR = \frac{Number\,of\,code\,smells\,needs\,to\,be\,removed}{Total number\,of\,code\,smells\,present\,in\,software} \tag{5.4}$$

where EE is defined as a ratio of total number of classes that needs to be refactored to the total number of classes present in the system. The formula to calculate EE is given in Eq. (5.5).

$$EE = \frac{Number\,of\,classes\,needs\,to\,be\,removed}{Total number\,of\,classes\,present\,in\,system} \tag{5.5}$$

$SFCS$ calculates the total severity of fixed code smells in software. It is calculated using Eq. (5.2.2).

$$SFCS = \sum_{i=1}^{n} TotalSeverity(A) \tag{5.6}$$

where n is total number of classes and $TotalSeverity(A)$ is calculated using Eq. (5.3).

5.3 Results and Discussions

This section represents and discusses the results obtained from three systems after implementing the proposed approach. Firstly, the findings acquired after analysing the previous versions of softwares are represented. Secondly, the results obtained after gathering the architectural relevant classes from current version of each dataset are discussed. Later, the results obtained after generating a rank of code smells are presented. Finally, the results of performance evaluation metrics employed on proposed approach are discussed.

- Analysis of Versioning History (VH)

  It is observed from Table 5.2 that in all the systems, the number of classes, KLOCs and code smells are increasing in their each new version. Figs. 5.4 to 5.6 reveal that between any two subsequent versions, the percentage of

changed classes is always greater than zero which is quite common for any software system in real world. It is observed that the percentage of changed classes for Log4j varies from 3% to 27%, which means its evolution trend is inconsistent as compared to other sample applications. The percentage of changed classes in Xerces-J is greater than 30% which is the greatest of all datasets. Similarly, the overall percentage of changed classes for GanttProject is 23%.

Furthermore, it is also analysed that in all the systems, the trend in percentage of changed classes is declining as we are moving towards current version. One of the main reason behind this trend may be that due to limited budget and time constraint the developers spent less efforts on applying refactoring activities.

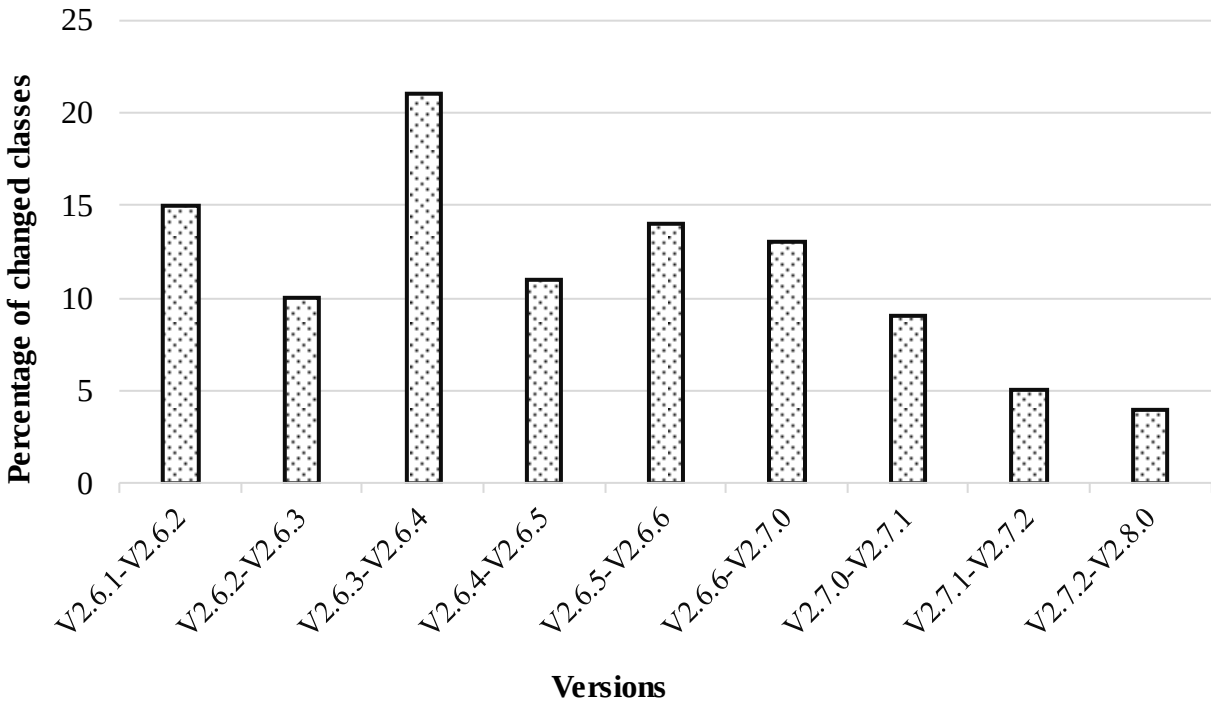


Fig. 5.4. Changed classes v/s succeeding version pairs of GanttProject.

- Analysis of Architecturally Relevant Classes (AR)

  The classes that are dangerous from architectural perspective of system are collected by analysing the current version of software. Fig. 5.7 reveals that

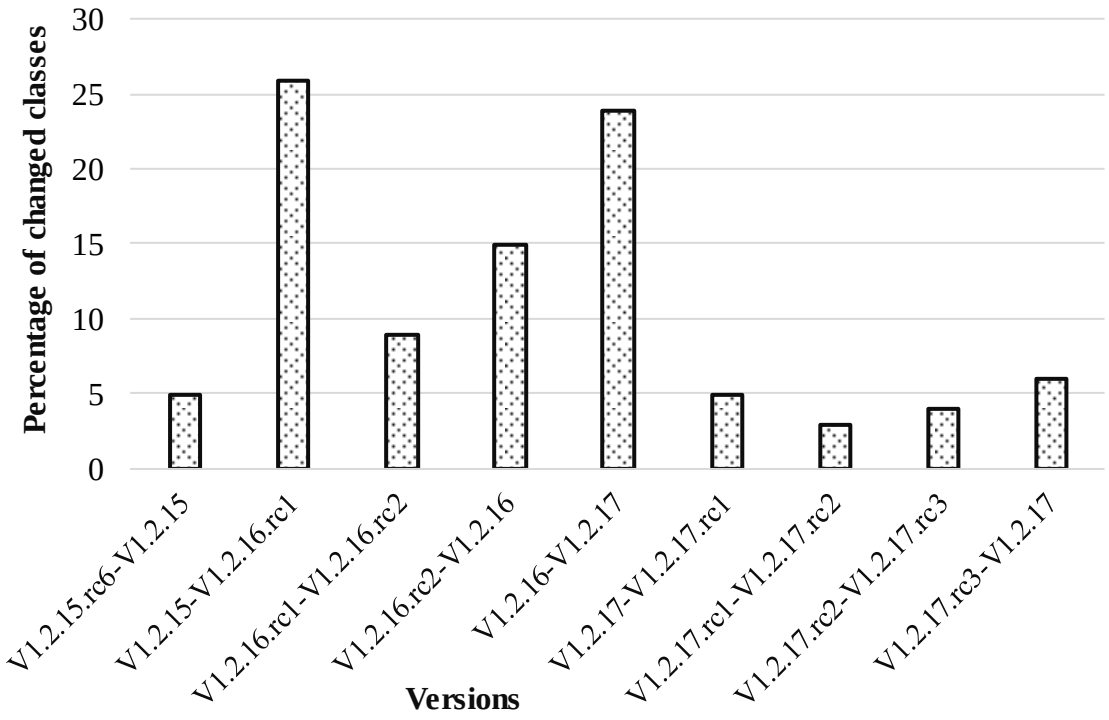


Fig. 5.5. Changed classes v/s succeeding version pairs of Log4j.

on average 27% of classes are architecturally relevant in all the systems. In other words, nearly 27% of classes are critical in current version of a system and needs immediate refactoring.

It is observed that for GanttProject out of 27% classes, 17% of classes are refactored in previous version. Whereas, in case of Log4j, 22% of classes out of 32% architecturally relevant classes, are refactored more than once in past. Similarly, for Xerces-J, 19% of classes out of 24% are refactored frequently in previous versions. Thus, it recognises that on average 21% classes out of those 27% classes are frequently refactored in the past and are chosen as most change-prone and significant classes. These classes are chosen on the fact that regardless of refactored in the earlier versions, such classes still have code smells and are more likely to change in future also. Therefore, these classes must be refactored at the earliest.

- Generation of Rank/Code Smell Score (RG)

  After selecting the common classes among frequently changed and

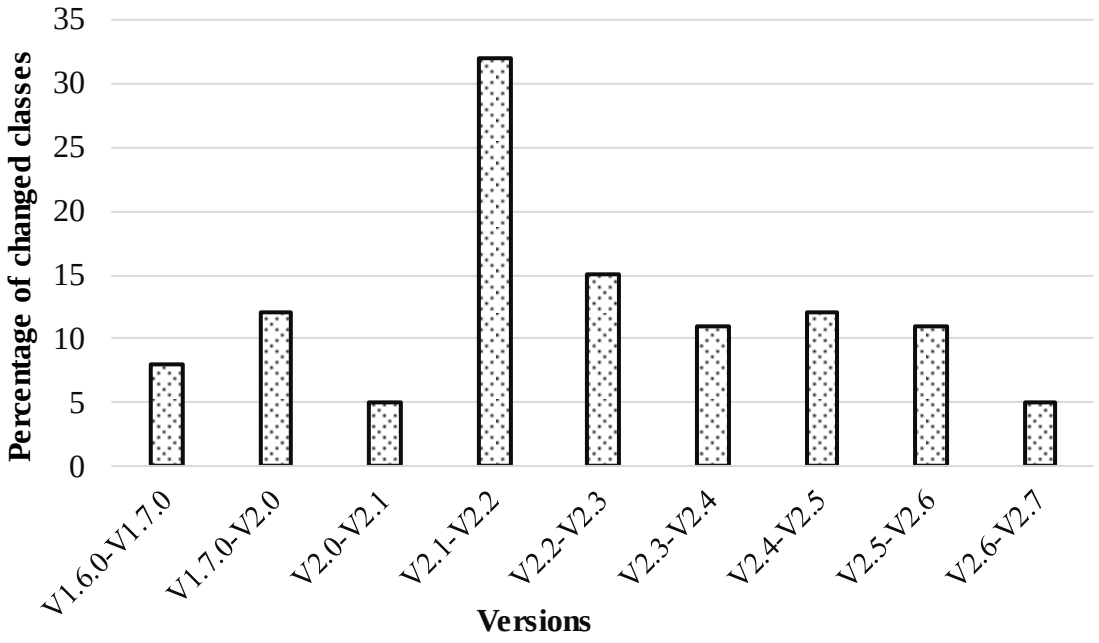


Fig. 5.6. Graph showing percentage of changed classes v/s succeeding version pairs of Xerces-J.

architecturally relevant classes, the severity to each code smell present in those classes is assigned. Furthermore, these three aforementioned parameters are given as an input to the proposed prioritization approach in order to generate code smell rank. Table 5.3 demonstrates top three high priority code smell and classes of each considered software system at $x = 0.1$ value. It is observed from Table 5.3 that the three parameters of code smell score contributed equally in generating ranks for the code smells. For example, for GanttProject, GanttGraphicPrint class has higher severity score than GanttGraphicArea or GanttOptions or GanttTree class; but it has lower number of code smell instances and smell relevance, therefore, it obtained a low code smell score (or rank).

In GanttProject, Feature Envy code smell has highest rank followed by Blob of two classes. Moreover, in Log4j, Blob code smell has highest priority followed by Feature Envy at second place and Data Class at third place, respectively. In addition, in Xerces-J, Feature Envy code smell scored first rank with Data

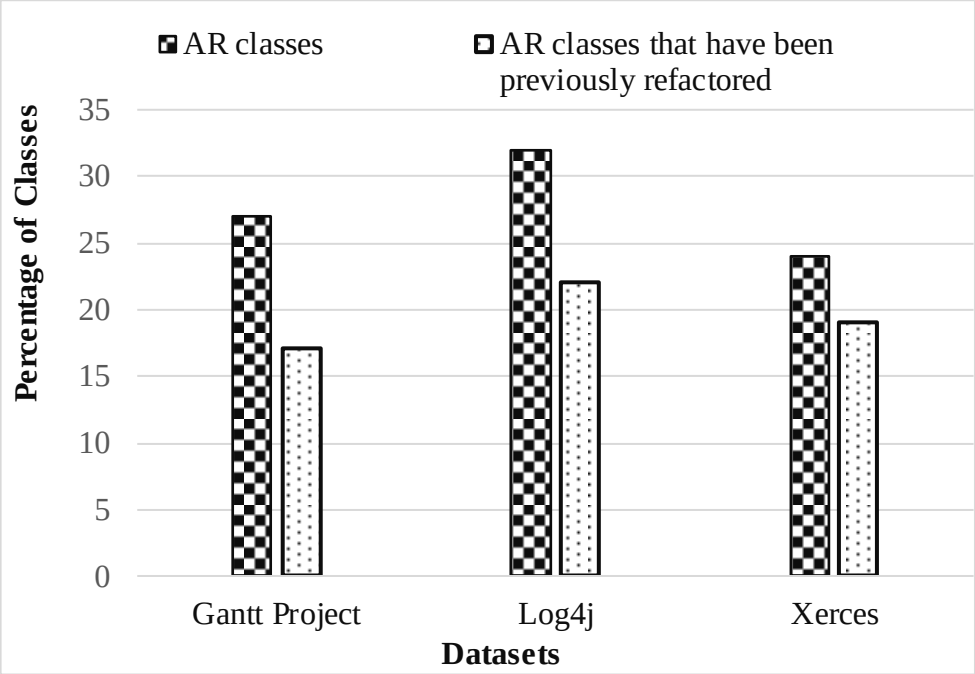


Fig. 5.7. Results on analyzing architecturally relevant classes.

Table 5.3: Top three high priority code smell and classes of each considered software system.

Datasets	Rank	Code Smell	Class Name	VH	CSR	AR	Code Smell Score
GanttProject	1	Feature Envy	GanttTree	5	0.7	17	15.65
	2	Blob	GanttGraphicArea	2	0.8	14	12.76
	3	Blob	GanttOptions	3	0.8	12	11.04
Log4j	1	Blob	Charsets class	3	0.8	15	13.74
	2	Feature Envy	AbstractLogger	2	0.7	11	10.04
	3	Data Class	MessageFactory	4	0.3	11	10.02
Xerces-J	1	Feature Envy	org.apache.xerces.impl	2	0.7	17	15.44
	2	Data Class	javax.xml.parsers.SAXParser	4	0.3	12	10.86
	3	Spaghetti Code	TeeXMLDocumentFilterImpl	5	0.6	6	5.7

Class at second rank and Spaghetti Code at third position, respectively. Thus, it is observed that the classes with more refactoring past history have higher number of code smell instances too. Hence, it is discovered that regardless

of refactored in the earlier versions of classes, such classes still have code smells and requires future attention.

- Evaluation Metrics

  This part represents the calculated results of proposed approach after using three performance metrics: CSCR, EE and SFCS. Evaluating these performance metrics at each stage further ensures that our approach is effective and efficient. Table 5.4 tabulated the results of proposed method for considered performance metrics.

  The table portrays that at first step **(VH)**, score of EE specifies on average, approximately 35% of the total classes needs refactoring. On other side, the results of CSCR shows that within these 35% classes, more than 78 % of code smells lies. Whereas, the results of AR describes that total of 18% classes are architecturally relevant. Thus, by applying the refactoring identification to these classes a substantial enhancement in the quality of a software is assured (as shown by CSCR value).

  It is also analysed from second step **(CSR)**, that the code smells that are refactored more in the earlier versions of a system are more relevant. Consequently, it is also observed at third step **(AR)**, on selected datasets nearly half of the earlier refactored code smells do not put any vulnerability to the architectural degradation. The results for Log4j reveals that nearly 56% (62 out of 109) classes are architecturally relevant. Accordingly, EE value for Log4j dataset is dropped to 21% with a nominal decline in CSRR value. Similar outcomes are obtained for GanttProject; an estimated refactoring effort of 18% and code smell correction of 75%. Consequently, EE and CSCR values for Xerces-J are dropped with a small decline of 19% and 64%, respectively. Thus, the classes with threshold value 20 are chosen. The developers can choose any value of x that supports them to make an equilibrium between CSRR, EE, and CSR value for their systems.

Table 5.4: Results obtained for performance metrics at each step for the considered dataset.

Dataset	Fitness Parameter	EE	CSCR	SFCS
GanttProject	VH	35% (183|532)	78% (523|671)	30.2
	CSR	22% (117|532)	76% (509|671)	32.9
	AR	18% (96|532)	75% (503|671)	29.2
	RG	15% (80|532)	70% (469|671)	31.6
Log4j	VH	37% (109|296)	75% (144|192)	24.6
	CSR	30% (89|296)	72% (138|192)	29.3
	AR	21% (62|296)	70% (134|192)	21.2
	RG	18% (53|296)	68% (130|192)	26.8
Xerces-J	VH	41% (210|513)	70% (228|325)	26.3
	CSR	36% (185|513)	68% (221|325)	31.7
	AR	19% (97|513)	64% (208|325)	24.2
	RG	16% (82|513)	60% (195|325)	29.4

5.4 Impact of Prioritization of Code Smells on Quality

In this section, the impact of code smell prioritization on internal software quality measures is investigated. Internal quality measures can be objectively measured from the source code of the software. Six C&K [20] and KLOC metrics are chosen to analyse the impact of prioritization of code smells on quality of software attributes. These metrics are chosen either due to the popularity of C&K quality measures [101], incorporation of these measures in most of the automated metric measurement tools, or because these measures are considered as benchmark for measuring quality. These metrics cover several software aspects including cohesion, coupling, complexity, inheritance and size. To measure these metrics, an Eclipse plug-in called metric suite is used. The detail of these considered metrics

is given in Table 5.5.

Table 5.5: Considered software metrics.

Quality attributes	**Measures**	**Description**
Cohesion	Lack of cohesion in methods (LCOM)	For each instance variable calculate the percentage of methods using it, then the average percentage for all variables subtracted from 100%.
Coupling	Coupling between object classes (CBO)	It is a count of the number of other classes to which a given class is coupled and, hence, denotes the dependency of one class on other classes in the design.
	Response for a Class (RFC)	This is the count of the methods that can be potentially invoked in response to a message received by an object of a particular class.
Complexity	Weighted methods per class (WMC)	This is a weighted sum of all the methods defined in a class.
Inheritance	Depth of Inheritance Tree (DIT)	The maximum length from the root class to a given class in the inheritance hierarchy.
	(Number Of Children (NOC)	The number of all child or subclasses that have inherited from a given class.
Size	Thousand Lines of Code (KLOC)	The number of thousand lines of code of the system.

To assess the effect of code smell prioritization on internal quality attributes, the metric values are collected corresponding each attribute for the three versions of the three software systems namely, Xerces-J, Log4j and GanttProject. The three software versions include the original version, the version generated by removing the code smells in random order and the version produced by eradicating the smells in prioritized order returned by our proposed approach. After calculating the metric values, these values are compared with each other to quantify the final effect of code smell prioritization on internal software quality. The detailed process of this methodology is depicted in Fig. 5.8.

5.4.1 Analysis of Code Smell Impact

The focus of this chapter is to validate/invalidate the claims that removal of code smells in the order generated by employing our approach improves software quality. To carry out this study, we set up five below-mentioned research questions.

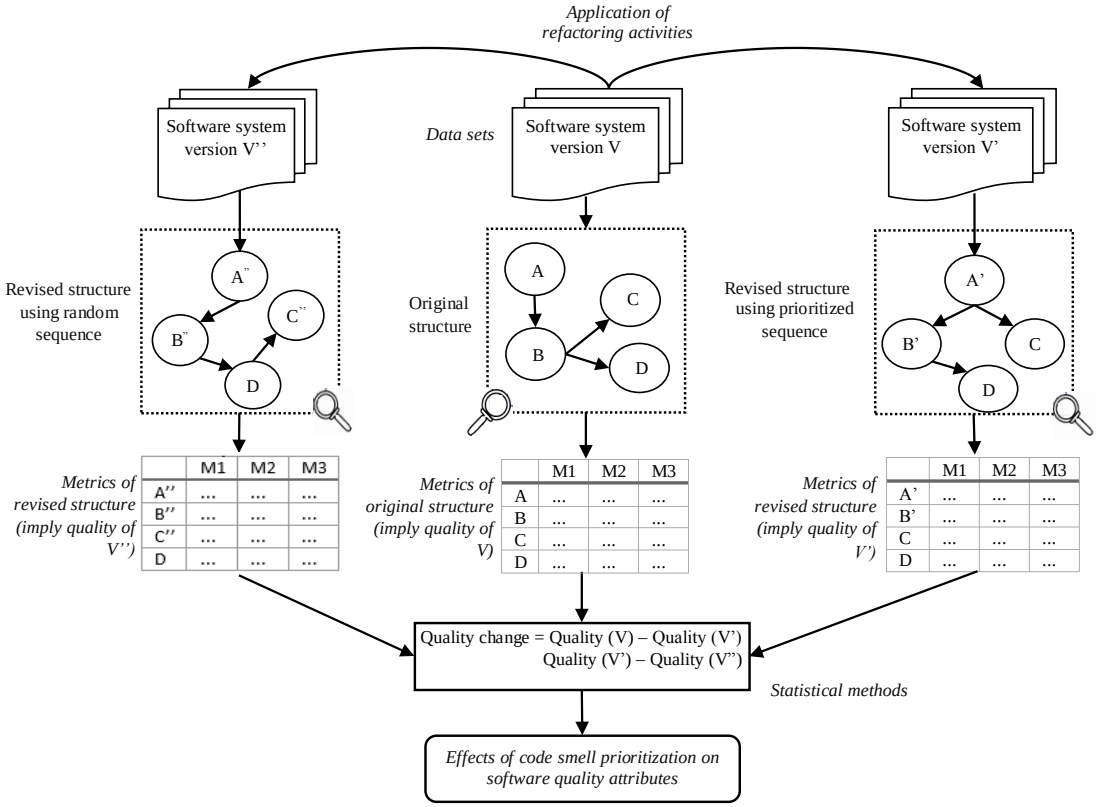


Fig. 5.8. Process flow of impact of code smells on software quality.

RQ1: Does the correction of code smells in prioritized order improves the software cohesion as compared to their correction in random order?

RQ2: Does the eradication of code smells in prioritized order have positive impact on software coupling as compared to their removal in random order?

RQ3: Does the correction of code smells in prioritized order improves the software size as compared to their correction in random order?

RQ4: Does the eradication of code smells in prioritized order have positive impact on software complexity as compared to their removal in random order?

RQ5: Does the correction of code smells in prioritized order improves the software inheritance as compared to their correction in random order?

Case Study: For every considered subject system, the impact of metric changes is examined following the application of refactoring activities to remove code smells. To answer RQ1-RQ5, the original version (V_1) of each considered dataset is refactored twice to produce its two new versions named V_2 and V_3. In V_2, code

smells are eradicated from V_1 in random order employing Eclipse IDE[5]. The same IDE is chosen to produce V_3 after eliminating code smells in the order generated by our proposed approach. The objective behind producing two different versions of same dataset is to observe and report the difference in them in terms of quality and to validate or invalidate the claim that code smell prioritization generates the better version. After generating three different versions, the metric values are computed for each version using an Eclipse plug-in named Metrics[6]. The outcomes concerning change in metric values of each considered C&K quality measure (enlisted in Table 5.5) for every dataset are reported in Table 5.6.

In this case study, we expect all metric values to deteriorate as reduction in their values signifies positive improvement. For each dataset, the deterioration and improvement of the value of each metric results in positive and negative impact on quality, respectively. This positive or negative result is indicated using a "+" or "-" sign, respectively. The cases with no improvement or deterioration in metric value are represented with "=" sign. For Xerces-J, the application of refactoring

Table 5.6: Impact of code smell prioritization on quality.

Datasets	**Version**	**Cohesion**	**Coupling**		**Complexity**	**Inheritance**		**Size**
		LCOM	CBO	RFC	WMC	DIT	NOC	LOC
Xerces-J	V_1-V_2	=	+0.38%	+0.17%	=	=	=	+0.10%
	V_1-V_3	+6.30%	+0.65%	+0.24%	+0.27%	=	=	+0.35%
Log4j	V_1-V_2	=	-3.84%	+7.65%	=	=	=	+1.34%
	V_1-V_3	=	+0.03%	+9.67%	=	=	=	+8.41%
GanttProject	V_1-V_2	+14.03%	-0.08%	-0.91%	+0.26%	=	=	-3.67%
	V_1-V_3	+32.34%	+0.04%	+0.53%	+0.41%	=	=	+25%

activities to remove code smells in the order generated by our proposed approach decreases the LOC, RFC, CBO by 0.35%, 0.24% and 0.65%, respectively. On the contrary, only 0.10%, 0.17% and 0.38% significant reduction are achieved by eliminating the smells in a random order for LOC, WMC, CBO, respectively. In addition, the version V_2 does not result in any change in cohesion and complexity

[5]https://www.eclipse.org/downloads/
[6]https://marketplace.eclipse.org/content/eclipse-metrics

whereas these quality attributes are improved significantly by employing our approach. The inheritance remains unchanged in both the versions V_2 and V_3.

For Log4j, the cohesion, complexity and inheritance remained unchanged. Further to that, the size and coupling of the software system gets reduced significantly. Somewhat different trend has been observed in GanttProject as the version V_2 caused size and coupling to increase whereas our approach reduced them significantly. Moreover, comparatively better results are achieved for other quality attributes. Overall, it is concluded that the removal of code smells in prioritized order improves the quality of Xerces-J, Log4j, and GanttProject. Thus, code smell prioritization assists the developers in producing better software and also save their effort and time thereby further enhances their productivity.

5.4.2 Statistical Testing

To statistically analyze the impact of code smell prioritization, a pair-wise t-test is performed. This statistical test usually applied in cases where samples are analysed in terms of changes as a result of exposure to a certain technique or environment. In addition, it does not make any assumptions regarding the distributions of the metrics The pair-wise t-test is beneficial in this thesis as it is able to identify the differences in quality metrics as result of refactoring. Using this test, the goal is to test following hypothesis.

Hypothesis: The removal of code bad smells in prioritized order improves the software quality than their removal in random order.

Null Hypothesis (H0): There is no difference between the quality of software versions V_1 and V_2.

Alternate Hypothesis (H1): There is a difference between the quality of software versions V_1 and V_2.

The null hypothesis is rejected in the case where quality metrics values after refactoring code smells in a random order are not equal to quality metrics values after refactoring the smells in a prioritized order generated by our proposed

approach. In this empirical study, we expect quality metric values of version V_2 to decrease to draw a conclusion on quality improvement in the considered projects. The quantification of the formulated hypothesis is necessary to later construct a judgement about the rejection or non-rejection of the null hypothesis. The quantification of our hypothesis is presented in terms of p-value as follow: Null Hypothesis (H0): p-values > 0.05 Alternate Hypothesis (H1): p-values < 0.05

Along with the considered hypothesis, C&K metrics and software quality are the independent and dependent variables, respectively. At 95% confidence level, the p-values of each internal quality metric generated from pair-wise t-test are shown in Table 5.7. It can be observed from this table that the p-values of all C&K metrics are less than 0.05 which implies that the metric values of software version V_3 generated after refactoring the code smells in prioritized order are statistically significant than the values of V_2. Thus, considering p-values of quality metrics, we reject the null hypothesis (H0) with 95% confidence. This aforementioned observation proves that code smells prioritization improves the quality and hence, these smells should be prioritized to reduce the developer's effort and save more time.

Table 5.7: $p-$values of C&K metrics at 95% confidence level.

Datasets	**Cohesion**	**Coupling**		**Complexity**	**Inheritance**		**Size**
	LCOM	CBO	RFC	WMC	DIT	NOC	LOC
Xerces-J	0.0227	0.0267	0.0012	0.0024	0.0070	0.0055	0.0329
Log4j	0.0136	0.0154	0.0028	0.0037	0.0021	0.0018	0.0487
GanttProject	0.0182	0.0273	0.0016	0.0025	0.0042	0.0037	0.0245

5.5 Summary

In this chapter, a novel code smell prioritization approach is proposed which ranks the detected code smells based on three parameters namely versioning history, code smell relevance and architectural relevance. By combining these parameters,

a code smell score is generated which further assists in ranking the considered code smells. The experimental results reveal that the proposed approach is more efficient towards prioritizing the code smells. In addition, the impact of code smell prioritization is also studied on cohesion, coupling, complexity, size, and inheritance attributes by performing a case study as well as statistical testing. The empirical evaluation proved that the eradication of smells as per their ranks produces a better quality software.

Chapter6

Conclusions and Future Scope

"Everything should be made as simple as possible, but not simpler." By Albert Einstein

In this thesis, firstly, the fundamental concepts of code smells, code smell detection process, need of code smell detection and code smell prioritization were discussed. Later, a thorough review of literature regarding code smell detection techniques has been done. This review was carried out on the basis of three aspects namely, code smells detection using machine-learning techniques, metaheuristic techniques and hybrid of both machine-learning and metaheuristic techniques. Also, the-state-of-the-art in the field of code smell prioritization was presented. Moreover, this thesis has presented a novel code smell detection approach using J48 decision tree and SPOA algorithm. Furthermore, SPOA has been also used with C5.0 algorithm called "C5.0-SPOA" for code smell detection. Also, code smells prioritization approach along with a process analysing the impact of code smells on software quality attributes was presented. Finally, section 6.1 concludes the thesis and section 6.2 discusses the future scope of work.

6.1 Conclusions

Code smells also called design defects, code anomalies or bad smells, are referred to the symptoms of design issues which can further hinder the maintenance of software system. Code smell handling is a major problem as these smells affect the quality of software. These smells are unlikely to cause failure directly, but may

do it indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. These code smell arises due to time and budget pressure as developers try to fix the maintenance problems in a limited time. These use shortcut methods or copy the code to other sections which leads to occurrence of code smells. Moreover, the presence of these code smells can also increase the maintenance cost, efforts, fault-proneness, defect-proneness, change-proneness and decrease the comprehensibility and understandability of a project. Therefore, the eradication of these code smells is very important.

The researchers have proposed large number of techniques to identify, prioritize and refactor these code smells according to their harmful impact on software quality. However, these existing techniques still suffer from several limitations. Therefore, to identify their shortcomings, a review of proposed approaches was carried out in the field of code smell detection and code smell prioritization. Furthermore, the comparison among these approaches was carried out and their shortcoming were highlighted as discussed in Section 2.4 of Chapter 2.

Based on identified research gaps, an efficient technique for code smell detection was proposed, as described in Chapters 3 and 4. In this approach, decision tree algorithm J48 was used to detect the code smells from Java OSS. In addition, code smell examples were used to generate detection rules for J48. The performance of the J48 has been evaluated on three considered datasets. The obtained results proved that j48 was more efficient than other existing techniques as it identified more code smells with 85% Precision and 77% Recall. But, it suffers from the problem of automatically selecting the most significant set of metrics while generating a decision tree. Therefore, to solve this issue J48 was hybridised with SPOA algorithm and named as "SP-J48".

SPOA is based on seabirds called sandpipers, which can be found all over the planet. The most important thing about the sandpipers is their searching and attacking behaviors. They can make their natural round shape movement

during attacking. These behaviors can be formulated in such a way that it can be associated with the objective function to be optimized. These features motivated us to towards formulating a new optimization algorithm. Further, the hybridisation of the proposed SPOA with J48 proved to be more efficient in detecting code smells with 88% Precision, 79% Recall, and 83% $F_{measure}$ in comparison to J48 and other existing techniques.

Furthermore, considering the advantages of C5.0 over J48, a new hybrid approach called "C5.0-SPOA" was proposed to detect code smell from Java OSS. It can be seen that C5.0 was fast, generate smaller trees and provides more accurate results comparatively J48. The performance of C5.0-SPOA was evaluated on the basis of three performance metrics. The obtained findings concluded that it outperforms other existing techniques with 91% Precision, 79% Recall, and 83% $F_{measure}$.

In third objective of this work, described in Chapter 5, the proposed SPOA approach was further extended to prioritize code smells for refactoring. The proposed technique supports the prioritization of most critical code smells that need immediate refactoring. Hence, the code smell prioritization was formulated as an optimization problem to find the near optimal sequence of code smells according to their critical effect.

A novel metaheuristic optimization approach was used to find the most critical code smells sequences while prioritizing the most important, architectural relevant and severe code smells according to developer's preferences. The performance of proposed approach was evaluated using three metrics namely, Estimated Effort, Code Smell Correction ratio, and Severity of Fixed Code Smells.

An average of 38% of the total classes in each application were frequently refactored in the previous versions. Out of those frequently refactored classes, almost half of the classes were architecturally relevant. On an average, 19% of the total classes in the current version for each sample application were architecturally relevant. Nearly 85% of refactoring effort can be saved by prioritizing the classes

based on the proposed approach and an average of 65% code smells gets corrected with the proposed class prioritization approach. In addition, the severity of every version was reduced with an average of 20 severity score. Overall, it was concluded from the reported results that the proposed approach was able to reduce developers effort by finding the severe code smells which in turn will save developer's time and maintenance cost.

Furthermore, the impact of code smells prioritisation on software internal quality attributes have been analyzed. It can be analyzed from previous studies that the code smell elimination has impact on internal quality attributes like cohesion, coupling, complexity, size and inheritance. In this analysis, it has been verified by analysing the impact of prioritization of five code smells on three Java OSS namely, GanttProject, Log4j and Xerces-J. The impact was analyzed by considering the three software versions: original version (V1), code smells removed in random order (V2) and code smells removed in prioritized order given by proposed prioritization approach (V3). These changes were analyzed in terms of C&K metrics of all the versions to identify the relationship for the same with cohesion, coupling, complexity, size and inheritance values.

Results of this analysis portrayed that for Xerces-J, code smells removed in the order generated by our proposed approach decreases the values of LOC, RFC, CBO metrics by 0.35%, 0.24%, and 0.65%, respectively. Whereas, with random sequence, only 0.10%, 0.17%, and 0.35% reduction was achieved. In addition, version V2 did not result in any change in cohesion and complexity whereas, these quality attributes were significantly improved by employing our prioritization approach. The inheritance attribute remained unchanged in both V2 and V3 versions. For Log4j, cohesion, complexity and inheritance remained unchanged whereas, size and complexity of system got significantly reduced by our proposed approach. For GanttProject, value of size and coupling got increased in version V2 but reduced significantly by our proposed approach. Moreover, a pair-wise t-test was performed to analyse the difference in quality metrics as a result of

refactoring.

At 5% level of significance, it was concluded that code smells refactored in prioritized order were statistically significant than the values of V2. Overall, we concluded that the removal of code smells in prioritized order improved the quality of Xerces-J, Log4j and GanttProject. Thus, code smell prioritization assists the developers in producing better software and also save their effort and time thereby further enhances their productivity.

Overall, this thesis can help the researchers and practitioners to understand preliminaries of code smells, code smell detection process, reasons of code smell occurrences, code smell disadvantages, need of code smell detection, code smell prioritization and its need according to their harmful impact on software quality, applications of machine-learning and meta-heuristic techniques in code smell detection and prioritization.

In addition, the researchers can have a deep insight regarding the impact of code smells on software quality before and after refactoring the code smells.

6.2 Future Scope

This research work achieved promising results with the proposed code smell detection and prioritization approach. However, some challenges were identified while carrying out this research work which could be considered for the future work. These challenges are:

- An important future direction consists of adapting our approach to work dynamically, i.e., metrics can be identified dynamically while the developers are analysing the metrics values.
- Detection of code smells can be enhanced by using the knowledge from the history of software changes. As reported in literature, classes involved in design problems (e.g. code smells) are much more likely to change. For these

reasons, the combination of static software metrics with historical software metrics can be an effective manner to enhance the detection of code-smells.

- The accuracy/performance of the proposed approach was carried out only on a limited set of sample applications. Thus, a large set of applications can be considered for further ensuring the accuracy of the results.
- The study included open source Java applications for evaluating the proposed approach. Hence, for generalization of the results, various commercial and industrial applications can be used. Along with industrial applications, the applications written in non-Java languages can be considered in future.
- The inclusion of feedback mechanism can be added in the algorithm, where the developer can give feedback on the detection to improve the learning model.
- More hybrid code smell detection techniques can be proposed to gain better results.
- For code-smells detection, the performance of our approach depends on the availability of code-smell examples, which could be difficult to collect. So to consider more programming contexts, the base examples can be extended by using additional badly-designed code.

AppendixA

Benchmark Test Functions

This appendix provides the details of single-objective benchmark test functions which are used in this thesis.

A.1 Unimodal Benchmark Test Functions

The detail description of seven well-known unimodal benchmark test functions ($F_1 - F_7$) is mentioned in Table A.1.

Table A.1: Unimodal benchmark test functions.

Function name	Mathematical formulation	Dim	Range	f_{min}
Sphere Model	$F_1(\vec{z}) = \sum_{i=1}^{N} z_i^2$	30	[-100, 100]	0
Schwefel's Problem 2.22	$F_2(\vec{z}) = \sum_{i=1}^{N} \|z_i\| + \prod_{i=1}^{N} \|z_i\|$	30	[-10, 10]	0
Schwefel's Problem 1.2	$F_3(\vec{z}) = \sum_{i=1}^{N} (\sum_{j-1}^{i} z_j)^2$	30	[-100, 100]	0
Schwefel's Problem 2.21	$F_4(\vec{z}) = max_i\{\|z_i\|, 1 \leq i \leq N\}$	30	[-100, 100]	0
Generalized Rosenbrock's Function	$F_5(\vec{z}) = \sum_{i=1}^{N-1} [100(z_{i+1} - z_i^2)^2 + (z_i - 1)^2]$	30	[-30, 30]	0
Step Function	$F_6(\vec{z}) = \sum_{i=1}^{N} (\|z_i + 0.5\|)^2$	30	[-100, 100]	0
Quartic Function	$F_7(\vec{z}) = \sum_{i=1}^{N} i z_i^4 + random[0,1]$	30	[-1.28, 1.28]	0

A.2 Multimodal Benchmark Test Functions

The detail description of six well-known multimodal benchmark test functions ($F_8 - F_{13}$) is mentioned in Table A.2.

Table A.2: Multimodal benchmark test functions.

Function name	Mathematical formulation	Dim	Range	f_{min}
Generalized Schwefel's Problem 2.26	$F_8(\vec{z}) = \sum_{i=1}^{N} -z_i sin(\sqrt{\lvert z_i \rvert})$	30	[-500, 500]	-12569.5
Generalized Rastrigin's Function	$F_9(\vec{z}) = \sum_{i=1}^{N} [z_i^2 - 10cos(2\pi z_i) + 10]$	30	[-5.12, 5.12]	0
Ackley's Function	$F_{10}(\vec{z}) = -20exp\left(-0.2\sqrt{\frac{1}{N}\sum_{i=1}^{N} z_i^2}\right) - exp\left(\frac{1}{N}\sum_{i=1}^{N} cos(2\pi z_i)\right) + 20 + e$	30	[-32, 32]	0
Generalized Griewank Function	$F_{11}(\vec{z}) = \frac{1}{4000}\sum_{i=1}^{N} z_i^2 - \prod_{i=1}^{N} cos\left(\frac{z_i}{\sqrt{i}}\right) + 1$	30	[-600, 600]	0
Generalized Penalized Functions	$F_{12}(\vec{z}) = \frac{\pi}{N}\{10sin(\pi x_1) + \sum_{i=1}^{N-1}(x_i - 1)^2[1 + 10sin^2(\pi x_{i+1})] + (x_n - 1)^2\} + \sum_{i=1}^{N} u(z_i, 10, 100, 4)$ $x_i = 1 + \frac{z_i + 1}{4}$ $u(z_i, a, k, m) = \begin{cases} k(z_i - a)^m & z_i > a \\ 0 & -a < z_i < a \\ k(-z_i - a)^m & z_i < -a \end{cases}$	30	[-50, 50]	0
	$F_{13}(\vec{z}) = 0.1\{sin^2(3\pi z_1) + \sum_{i=1}^{N}(z_i - 1)^2[1 + sin^2(3\pi z_i + 1)] + (z_n - 1)^2[1 + sin^2(2\pi z_n)]\} + \sum_{i=1}^{N} u(z_i, 5, 100, 4)$	30	[-50, 50]	0

A.3 Fixed-dimension Multimodal Benchmark Test Functions

The detail description of ten well-known fixed-dimension multimodal benchmark test functions ($F_{14} - F_{23}$) is mentioned in Table A.3.

Table A.3: Fixed-dimension multimodal benchmark test functions.

Function name	Mathematical formulation	Dim	Range	f_{min}
Shekel's Foxholes Function	$F_{14}(\vec{z}) = \left(\frac{1}{500} + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2}(z_i - a_{ij})^6}\right)^{-1}$	2	[-65, 65]	1
Kowalik's Function	$F_{15}(\vec{z}) = \sum_{i=1}^{11} \left[a_i - \frac{z_1(b_i^2 + b_i z_2)}{b_i^2 + b_i z_3 + z_4}\right]^2$	4	[-5, 5]	0.00030
Six-Hump Camel-Back Function	$F_{16}(\vec{z}) = 4z_1^2 - 2.1z_1^4 + \frac{1}{3}z_1^6 + z_1 z_2 - 4z_2^2 + 4z_2^4$	2	[-5, 5]	-1.0316
Branin Function	$F_{17}(\vec{z}) = \left(z_2 - \frac{5.1}{4\pi^2}z_1^2 + \frac{5}{\pi}z_1 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)cos z_1 + 10$	2	[-5, 5]	0.398
Goldstein-Price Function	$F_{18}(\vec{z}) = [1 + (z_1 + z_2 + 1)^2(19 - 14z_1 + 3z_1^2 - 14z_2 + 6z_1 z_2 + 3z_2^2)] \times [30 + (2z_1 - 3z_2)^2 \times (18 - 32z_1 + 12z_1^2 + 48z_2 - 36z_1 z_2 + 27z_2^2)]$	2	[-2, 2]	3
Hartman's Family	$F_{19}(\vec{z}) = -\sum_{i=1}^{4} c_i exp(-\sum_{j=1}^{3} a_{ij}(z_j - p_{ij})^2)$	3	[1, 3]	-3.86
	$F_{20}(\vec{z}) = -\sum_{i=1}^{4} c_i exp(-\sum_{j=1}^{6} a_{ij}(z_j - p_{ij})^2)$	6	[0, 1]	-3.32
Shekel's Foxholes Function	$F_{21}(\vec{z}) = -\sum_{i=1}^{5}[(X - a_i)(X - a_i)^T + c_i]^{-1}$	4	[0, 10]	-10.1532
	$F_{22}(\vec{z}) = -\sum_{i=1}^{7}[(X - a_i)(X - a_i)^T + c_i]^{-1}$	4	[0, 10]	-10.4028
	$F_{23}(z) = -\sum_{i=1}^{1} 0[(X - a_i)(X - a_i)^T + c_i]^{-1}$	4	[0, 10]	-10.536

A.4 IEEE CEC 2015 benchmark test functions

The detail description of fifteen well-known composite benchmark test functions ($CEC-1-CEC15$) is mentioned in Table A.4.

Table A.4: CEC 2015 benchmark test functions.

No.	Functions	Related basic functions	Dim	f_{min}
$CEC-1$	Rotated Bent Cigar Function	Bent Cigar Function	30	100
$CEC-2$	Rotated Discus Function	Discus Function	30	200
$CEC-3$	Shifted and Rotated Weierstrass Function	Weierstrass Function	30	300
$CEC-4$	Shifted and Rotated Schwefel's Function	Schwefel's Function	30	400
$CEC-5$	Shifted and Rotated Katsuura Function	Katsuura Function	30	500
$CEC-6$	Shifted and Rotated HappyCat Function	HappyCat Function	30	600
$CEC-7$	Shifted and Rotated HGBat Function	HGBat Function	30	700
$CEC-8$	Shifted and Rotated Expanded Griewank's plus Rosenbrock's Function	Griewank's Function Rosenbrock's Function	30	800
$CEC-9$	Shifted and Rotated Expanded Scaffer's F6 Function	Expanded Scaffer's F6 Function	30	900
$CEC-10$	Hybrid Function 1 ($N=3$)	Schwefel's Function Rastrigin's Function High Conditioned Elliptic Function	30	1000
$CEC-11$	Hybrid Function 2 ($N=4$)	Griewank's Function Weierstrass Function Rosenbrock's Function Scaffer's F6 Function	30	1100
$CEC-12$	Hybrid Function 3 ($N=5$)	Katsuura Function HappyCat Function Expanded Griewank's plus Rosenbrock's Function Schwefel's Function Ackley's Function	30	1200
$CEC-13$	Composition Function 1 ($N=5$)	Rosenbrock's Function High Conditioned Elliptic Function Bent Cigar Function Discus Function High Conditioned Elliptic Function	30	1300
$CEC-14$	Composition Function 2 ($N=3$)	Schwefel's Function Rastrigin's Function High Conditioned Elliptic Function	30	1400
$CEC-15$	Composition Function 3 ($N=5$)	HGBat Function Rastrigin's Function Schwefel's Function Weierstrass Function High Conditioned Elliptic Function	30	1500

References

[1] S. Mamone, "The ieee standard for software maintenance," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 1, pp. 75–76, 1994.

[2] S. S. Rathore and S. Kumar, "Towards an ensemble based system for predicting the number of software faults," *Expert Systems with Applications*, vol. 82, pp. 357–382, 2017.

[3] ——, "A study on software fault prediction techniques," *Artificial Intelligence Review*, vol. 51, no. 2, pp. 255–327, 2019.

[4] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976, pp. 492–497.

[5] N. Chapin, "Do we know what preventive maintenance is?" in *icsm*, 2000, pp. 15–17.

[6] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry, "Leveraging software product lines engineering in the development of external dsls: A systematic literature review," *Computer Languages, Systems & Structures*, vol. 46, pp. 206–235, 2016.

[7] I. Haikala and J. Märijärvi, "Ohjelmistotuotanto gummerus kirjapaino oy," 1998.

[8] R. L. Glass and R. A. Noiseux, *Software maintenance guidebook*. Prentice Hall, 1981.

[9] T. M. Pigoski, *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.

[10] T. DeMarco, *Controlling software projects: Management, measurement, and estimates*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1986.

[11] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[12] B. C. Wagey, B. Hendradjaya, and M. S. Mardiyanto, "A proposal of software maintainability model using code smell measurement," in *2015 International Conference on Data and Software Engineering (ICoDSE)*. IEEE, 2015, pp. 25–30.

[13] A. Yamashita, "How good are code smells for evaluating software maintainability? results from a comparative case study," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 566–571.

[14] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

[15] M. H. Halstead *et al.*, *Elements of software science*. Elsevier New York, 1977, vol. 7.

[16] V. Laing and C. Coleman, "Principal components of orthogonal object-oriented metrics," *Software Assurance Technology Center, White Paper SATC-323-08-14, NASA Goddard Space Flight Center, Greenbelt, Maryland*, vol. 20771, 2001.

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

[18] R. Jabangwe, J. Börstler, D. Šmite, and C. Wohlin, "Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review," *Empirical Software Engineering*, vol. 20, no. 3, pp. 640–693, 2015.

[19] D. Boshnakoska and A. Mišev, "Correlation between object-oriented metrics and refactoring," in *International Conference on ICT Innovations*. Springer, 2010, pp. 226–235.

[20] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[21] W. Li and S. Henry, "Maintenance metrics for the object oriented paradigm," in *[1993] Proceedings First International Software Metrics Symposium*. IEEE, 1993, pp. 52–60.

[22] F. B. Abreu and R. Carapuça, "Object-oriented software engineering: Measuring and controlling the development process," in *Proceedings of the 4th international conference on software quality*, vol. 186, 1994, pp. 1–8.

[23] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[24] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," 1991.

[25] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of systems and software*, vol. 23, no. 2, pp. 111–122, 1993.

[26] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment." *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[27] G. Booch, *Object oriented analysis & design with application*. Pearson Education India, 2006.

[28] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 403–414.

[29] M. Fowler, K. Beck, J. Brant, and W. Opdyke, "Refactoring: improving the design of existing code. 1999," *Cited on*, p. 12.

[30] W. Abdelmoez, E. Kosba, and A. F. Iesa, "Risk-based code smells detection tool," in *The International Conference on Computing Technology and Information Management (ICCTIM)*. Society of Digital Information and Wireless Communication, 2014, p. 148.

[31] W. F. Opdyke, "Refactoring object-oriented frameworks," 1992.

[32] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 350–359.

[33] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[34] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[35] S. R. Schach, *Object-oriented and classical software engineering*. McGraw-Hill New York, 2007, vol. 6.

[36] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE'04.* IEEE, 2004, pp. 83–92.

[37] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 55–64.

[38] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[39] J. Krinke, N. Gold, Y. Jia, and D. Binkley, "Cloning and copying between gnome projects," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 98–101.

[40] M. Mantyla, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, 2003, pp. 381–384.

[41] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015.

[42] M. Mantyla, "Bad smells in software-a taxonomy and an empirical study," *Helsinki University of Technology*, 2003.

[43] P. Atri, "Detect malodorous software pattern and refactor them," Ph.D. dissertation, Sciences, 2012.

[44] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, 2006.

[45] W. C. Wake, *Refactoring workbook*. Addison-Wesley Professional, 2004.

[46] N. Mathur, "Java smell detector," 2011.

[47] A. Sabané, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A study on the relation between antipatterns and the cost of class unit testing," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 167–176.

[48] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[49] S. Kumar and J. K. Chhabra, "Two level dynamic approach for feature envy detection," in *2014 International Conference on Computer and Communication Technology (ICCCT)*. IEEE, 2014, pp. 41–46.

[50] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos, "An empirical investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 65, no. 2, pp. 127–139, 2003.

[51] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 187–196.

[52] D. Bán and R. Ferenc, "Recognizing antipatterns and analyzing their effects on software maintainability," in *International Conference on Computational Science and Its Applications*. Springer, 2014, pp. 337–352.

[53] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," in *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No. 02CH37318)*. IEEE, 2002, pp. 235–241.

[54] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: research and practice*, vol. 23, no. 3, pp. 179–202, 2011.

[55] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[56] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 392–395.

[57] D. Rapu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* IEEE, 2004, pp. 223–232.

[58] R. Marinescu, G. Ganea, and I. Verebi, "Incode: Continuous quality assessment and improvement," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 274–275.

[59] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 278–281.

[60] G. Ganea, I. Verebi, and R. Marinescu, "Continuous quality assessment with incode," *Science of Computer Programming*, vol. 134, pp. 19–36, 2017.

[61] M. J. Munro, "Product metrics for automatic identification of" bad smell" design problems in java source-code," in *11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE, 2005, pp. 15–15.

[62] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[63] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *Proceedings of the Sixth international workshop on emerging trends in software metrics*. IEEE Press, 2015, pp. 44–53.

[64] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 612–621.

[65] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in neural information processing systems*, 2015, pp. 2503–2511.

[66] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[67] ——, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[68] N. SaravanaN and V. Gayathri, "Performance and classification evaluation of j48 algorithm and kendall's based j48 algorithm (knj48)."

[69] G. Kaur and A. Chhabra, "Improved j48 classification algorithm for the prediction of diabetes," *International Journal of Computer Applications*, vol. 98, no. 22, 2014.

[70] R. Pandya and J. Pandya, "C5. 0 algorithm to improved decision tree with feature selection and reduced error pruning," *International Journal of Computer Applications*, vol. 117, no. 16, pp. 18–21, 2015.

[71] N. Patil, R. Lathi, and V. Chitre, "Comparison of c5. 0 & cart classification algorithms using pruning technique," *Int. J. Eng. Res. Technol*, vol. 1, no. 4, pp. 1–5, 2012.

[72] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *ACM Sigplan Notices*, vol. 34, no. 10. ACM, 1999, pp. 47–56.

[73] M. Salehie, S. Li, and L. Tahvildari, "A metric-based heuristic framework to detect object-oriented design flaws," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 2006, pp. 159–168.

[74] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2013, pp. 116–125.

[75] Y. Cai, L. Xiao, R. Kazman, R. Mo, and Q. Feng, "Design rule spaces: A new model for representing and analyzing software architecture," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 657–682, 2018.

[76] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 977–1000, 2017.

[77] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[78] R. Marinescu and D. Ratiu, "Quantifying the quality of object-oriented design: The factor-strategy model," in *11th Working Conference on Reverse Engineering*. IEEE, 2004, pp. 192–201.

[79] A. A. Rao and K. N. Reddy, "Detecting bad smells in object oriented design using design change propagation probability matrix 1," 2007.

[80] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. IEEE, 2001, pp. 30–38.

[81] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the 5th international symposium on Software visualization*. ACM, 2010, pp. 5–14.

[82] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[83] V. Kadyan, A. Mantri, and R. Aggarwal, "A heterogeneous speech feature vectors generation approach with hybrid hmm classifiers," *International Journal of Speech Technology*, vol. 20, no. 4, pp. 761–769, 2017.

[84] A. Slowik and M. Bialko, "Training of artificial neural networks using differential evolution algorithm," in *2008 conference on human system interactions*. IEEE, 2008, pp. 60–65.

[85] J. Kreimer, "Adaptive detection of design flaws," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, 2005.

[86] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 145–154.

[87] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*. IEEE, 2009, pp. 305–314.

[88] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 248–251.

[89] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, and S. Hamel, "Ids: An immune-inspired approach for the detection of software design smells," in *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 2010, pp. 343–348.

[90] N. Maneerat and P. Muenchaisri, "Bad-smell prediction from software design model using machine learning techniques," in *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 2011, pp. 331–336.

[91] P. Danphitsanuphan and T. Suwantada, "Code smell detecting tool and code smell-structure bug relationship," in *2012 Spring Congress on Engineering and Technology*. IEEE, 2012, pp. 1–5.

[92] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 466–475.

[93] S. Fu and B. Shen, "Code bad smell detection through evolutionary data mining," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–9.

[94] *https://www.sonarqube.org/*, https://www.sonarqube.org/.

[95] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 268–278.

[96] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014.

[97] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code smell detection: Towards a machine learning-based approach," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 396–399.

[98] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Can i clone this piece of code here?" in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 170–179.

[99] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[100] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[101] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and software technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[102] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, and A. Egyed, "An assessment of search-based techniques for reverse engineering feature models," *Journal of Systems and Software*, vol. 103, pp. 353–369, 2015.

[103] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE, 2001, pp. 107–114.

[104] T. K. Landauer and S. T. Dumais, "A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge." *Psychological review*, vol. 104, no. 2, p. 211, 1997.

[105] P. F. Mihancea and R. Marinescu, "Towards the optimization of automatic detection of design flaws in object-oriented software systems," in *Ninth*

European conference on software maintenance and reengineering. IEEE, 2005, pp. 92–101.

[106] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, "Search-based design defects detection by example," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2011, pp. 401–415.

[107] Y. Shi *et al.*, "Particle swarm optimization: developments, applications and resources," in *Proceedings of the 2001 congress on evolutionary computation (IEEE Cat. No. 01TH8546)*, vol. 1. IEEE, 2001, pp. 81–86.

[108] K. S. Lee and Z. W. Geem, "A new meta-heuristic algorithm for continuous engineering optimization: harmony search theory and practice," *Computer methods in applied mechanics and engineering*, vol. 194, no. 36-38, pp. 3902–3933, 2005.

[109] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[110] U. Mansoor, M. Kessentini, S. Bechikh, and K. Deb, "Code-smells detection using good and bad software design examples," *Technical report, Technical Report*, 2013.

[111] M. Gong, L. Jiao, H. Du, and L. Bo, "Multiobjective immune algorithm with nondominated neighbor-based selection," *Evolutionary Computation*, vol. 16, no. 2, pp. 225–255, 2008.

[112] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 2011, pp. 81–90.

[113] M. Kessentini, S. Vaucher, and H. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in

Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, 2010, pp. 113–122.

[114] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, vol. 25, no. 2, pp. 529–552, 2017.

[115] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. B. Chikha, "Competitive coevolutionary code-smells detection," in *International Symposium on Search Based Software Engineering*. Springer, 2013, pp. 50–65.

[116] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[117] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 2008, pp. 329–331.

[118] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 1, p. 6, 2014.

[119] A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, vol. 24, no. 4, pp. 947–965, 2016.

[120] A. Ghannem, M. Kessentini, and G. El Boussaidi, "Detecting model refactoring opportunities using heuristic search," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 2011, pp. 175–187.

[121] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, "Experience report: Evaluating the effectiveness of decision trees for detecting code smells," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 261–269.

[122] C.-W. Hsu, C.-C. Chang, C.-J. Lin *et al.*, "A practical guide to support vector classification," 2003.

[123] G. Saranya, H. K. Nehemiah, A. Kannan, and V. Nithya, "Model level code smell detection using egapso based on similarity measures," *Alexandria engineering journal*, 2017.

[124] M. Hadj-Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning." in *ENASE*, 2018, pp. 137–146.

[125] Y. Bengio *et al.*, "Learning deep architectures for ai," *Foundations and trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[126] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[127] H. Liu, Q. Liu, Z. Niu, and Y. Liu, "Dynamic and automatic feedback-based threshold adaptation for code smell detection," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 544–558, 2015.

[128] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based approach to prioritize code smells for prefactoring," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1886, 2018.

[129] D. Steidl and S. Eder, "Prioritizing maintainability defects based on refactoring recommendations," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 168–176.

[130] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka, "Tool support for continuous quality control," *IEEE software*, vol. 25, no. 5, pp. 60–67, 2008.

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

[132] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, "Jspirit: a flexible tool for the analysis of code smells," in *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, 2015, pp. 1–6.

[133] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos, "Identifying architectural problems through prioritization of code smells," in *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. IEEE, 2016, pp. 41–50.

[134] M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Prioritising refactoring using code bad smells," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 458–464.

[135] *Pmd.sourceforge.net 2009, PMD, Available at: http://pmd.sourceforge.net/ [Accessed 15 March, 2019]*, Pmd.sourceforge.net2009,PMD,Availableat: http://pmd.sourceforge.net/[Accessed15March,2019].

[136] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Filtering clones for individual user based on machine learning analysis," in *Proceedings of the 6th International Workshop on Software Clones*. IEEE Press, 2012, pp. 76–77.

[137] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2015, pp. 16–24.

[138] V. Ferme, "Jcodeodor: A software quality advisor through design flaws detection," *Master's thesis, University of Milano-Bicocca, Milano, Italy*, 2013.

[139] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 336–345.

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

[141] E. Alba and B. Dorronsoro, "The exploration/exploitation tradeoff in dynamic cellular genetic algorithms," *IEEE transactions on evolutionary computation*, vol. 9, no. 2, pp. 126–142, 2005.

[142] O. Olorunda and A. P. Engelbrecht, "Measuring exploration/exploitation in particle swarms using swarm diversity," in *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE, 2008, pp. 1128–1134.

[143] M. Lozano and C. García-Martínez, "Hybrid metaheuristics with evolutionary algorithms specializing in intensification and diversification: Overview and progress report," *Computers & Operations Research*, vol. 37, no. 3, pp. 481–497, 2010.

[144] S. Kaur, L. K. Awasthi, A. Sangal, and G. Dhiman, "Tunicate swarm algorithm: A new bio-inspired based metaheuristic paradigm for global optimization," *Engineering Applications of Artificial Intelligence*, vol. 90, p. 103541, 2020.

[145] S. Kaur, L. K. Awasthi, and A. Sangal, "Hmoshssa: a hybrid meta-heuristic approach for solving constrained optimization problems," *Engineering with Computers*, pp. 1–37, 2020.

[146] R. K. Aggarwal and M. Dave, "Filterbank optimization for robust asr using ga and pso," *International Journal of Speech Technology*, vol. 15, no. 2, pp. 191–201, 2012.

[147] A. Shah and K. Kotecha, "Scheduling algorithm for real-time operating systems using aco," in *2010 International Conference on Computational Intelligence and Communication Networks*. IEEE, 2010, pp. 617–621.

[148] ——, "Aco based dynamic scheduling algorithm for real-time multiprocessor systems," *International Journal of Grid and High Performance Computing (IJGHPC)*, vol. 3, no. 3, pp. 20–30, 2011.

[149] H.-z. LI, S. GUO, and C.-j. LI, "A hybrid forecasting model based on fruit fly optimization algorithm and least squares support vector machine:——the case of logistics demand forecasting of china [j]," *Journal of Quantitative Economics*, vol. 3, 2012.

[150] L. Hongze, G. Sen, W. Bao *et al.*, "Evaluation on power customer value based on ants colony clustering algorithm optimized by genetic algorithm," *Power System Technology*, vol. 36, no. 12, pp. 256–261, 2012.

[151] A. R. Yildiz, "Cuckoo search algorithm for the selection of optimal machining parameters in milling operations," *The International Journal of Advanced Manufacturing Technology*, vol. 64, no. 1-4, pp. 55–61, 2013.

[152] ——, "A comparative study of population-based optimization algorithms for turning operations," *Information Sciences*, vol. 210, pp. 81–88, 2012.

[153] A. Slowik and H. Kwasnicka, "Nature inspired methods and their industry applications—swarm intelligence algorithms," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 3, pp. 1004–1015, 2017.

[154] J. Kennedy, "Particle swarm optimization," in *Encyclopedia of machine learning*. Springer, 2011, pp. 760–766.

[155] G. Dhiman and V. Kumar, "Spotted hyena optimizer: a novel bio-inspired based metaheuristic technique for engineering applications," *Advances in Engineering Software*, vol. 114, pp. 48–70, 2017.

[156] ——, "Multi-objective spotted hyena optimizer: A multi-objective optimization algorithm for engineering problems," *Knowledge-Based Systems*, vol. 150, pp. 175–197, 2018.

[157] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Grey wolf optimizer," *Advances in engineering software*, vol. 69, pp. 46–61, 2014.

[158] S. Mirjalili, "Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm," *Knowledge-Based Systems*, vol. 89, pp. 228–249, 2015.

[159] S. Mirjalili, S. M. Mirjalili, and A. Hatamlou, "Multi-verse optimizer: a nature-inspired algorithm for global optimization," *Neural Computing and Applications*, vol. 27, no. 2, pp. 495–513, 2016.

[160] S. Mirjalili, "Sca: a sine cosine algorithm for solving optimization problems," *Knowledge-Based Systems*, vol. 96, pp. 120–133, 2016.

[161] E. Rashedi, H. Nezamabadi-Pour, and S. Saryazdi, "Gsa: a gravitational search algorithm," *Information sciences*, vol. 179, no. 13, pp. 2232–2248, 2009.

[162] E. Bonabeau, D. d. R. D. F. Marco, M. Dorigo, G. Théraulaz, G. Theraulaz *et al.*, *Swarm intelligence: from natural to artificial systems*. Oxford university press, 1999, no. 1.

[163] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[164] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[165] Q. Chen, B. Liu, Q. Zhang, J. Liang, P. Suganthan, and B. Qu, "Problem definitions and evaluation criteria for cec 2015 special session on bound constrained single-objective computationally expensive numerical optimization," *Technical Report, Nanyang Technological University*, 2014.

[166] M. Jureczko and D. Spinellis, "Using object-oriented design metrics to predict software defects," *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej*, pp. 69–81, 2010.

[167] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," in *11th Working Conference on Reverse Engineering*. IEEE, 2004, pp. 172–181.

[168] N. Bhargava, G. Sharma, R. Bhargava, and M. Mathuria, "Decision tree analysis on j48 algorithm for data mining," *Proceedings of International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 6, 2013.

[169] D. Peng, F. C. Lee, and D. Boroyevich, "A novel svm algorithm for multilevel three-phase converters," in *2002 IEEE 33rd Annual IEEE Power Electronics Specialists Conference. Proceedings (Cat. No. 02CH37289)*, vol. 2. IEEE, 2002, pp. 509–513.

[170] J. R. Anderson and M. Matessa, "Explorations of an incremental, bayesian algorithm for categorization," *Machine Learning*, vol. 9, no. 4, pp. 275–308, 1992.

[171] P. Kapoor, R. Rani, and R. JMIT, "Efficient decision tree algorithm using j48 and reduced error pruning," *International Journal of Engineering Research and General Science*, vol. 3, no. 3, pp. 1613–1621, 2015.

[172] S. Taneja, "Implementation of novel algorithm (spruning algorithm)," *IOSR Journal of Computer Engineering (IOSR-JCE)*, pp. 57–65, 2014.

[173] T. Girba, S. Ducasse, and M. Lanza, "Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 40–49.

[174] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[175] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Software Quality Journal*, vol. 23, no. 2, pp. 323–361, 2015.

[176] R. Malhotra, A. Chug, and P. Khosla, "Prioritization of classes for refactoring: A step towards improvement in software quality," in *Proceedings of the Third International Symposium on Women in Computing and Informatics*, 2015, pp. 228–234.

[177] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of software maintenance and evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[178] A. G. Koru and H. Liu, "Identifying and characterizing change-prone classes in two large-scale open-source products," *Journal of Systems and Software*, vol. 80, no. 1, pp. 63–73, 2007.

[179] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó'Cinnéide, and K. Deb, "Software refactoring under uncertainty: a robust multi-objective approach," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 2014, pp. 187–188.

[180] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 255–258.

[181] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. V. Staa, "On the relationship of code-anomaly agglomerations and architectural problems," *Journal of Software Engineering Research and Development*, vol. 3, no. 1, p. 11, 2015.

Author's Publications

The contribution of author's accepted publications in this thesis are mentioned below:

1. A. Kaur, S. Jain and S. Goel. SP-J48: a novel optimization and machine-learning-based approach for solving complex problems: special application in software engineering for detecting code smells. *Neural Computing and Applications (Springer)*, 1–19, 2019. **[SCI/SCIE Indexed, Impact Factor - 4.664]**

2. A. Kaur, S. Jain and S. Goel. Sandpiper optimization algorithm: a novel approach for solving real-life engineering problems. *Applied Intelligence (Springer)*, 582–619, 2020. **[SCI/SCIE Indexed, Impact Factor - 2.882]**

3. A. Kaur, S. Jain and S. Goel. A support vector machine based approach for code smell detection. In *International Conference on Machine Learning and Data Science (MLDS)*, 2017, 9–14, *IEEE*. **[Web of Science Indexed]**

4. A. Kaur, S. Jain, S. Goel and G. Dhiman. A Review on Machine-Learning Based Code Smell Detection Techniques in Object-Oriented Software System(s). *Recent Advances in Electrical & Electronic Engineering (Bentham Science)*, 2020. **[ESCI Indexed]**

5. A. Kaur, S. Jain, S. Goel and G. Dhiman. Prioritization of Code Smells in Object-Oriented Software: A Review. *Materials Today: Proceedings (Elsevier)*, 2020. **[SCOPUS Indexed]**

The contribution of author's communicated publications in this thesis are mentioned below:

1. A. Kaur, S. Jain and S. Goel. A novel code smell prioritization technique to study the impact on software quality attributes. *Engineering with Computers (Springer)*, **[SCI/SCIE Indexed, Impact Factor - 3.938] (*Under Review*)**

2. A. Kaur, S. Jain and S. Goel. A novel hybrid code smell detection technique

using C5.0-SPOA. *Knowledge-Based Systems (Elsevier)*, **[SCI/SCIE Indexed, Impact Factor - 5.921] (*Under Review*)**

Index