Detection of Antipatterns from Service Oriented Architecture

By

Fatima Sabir

CIIT/SP14-PCS-003/LHR

PhD Thesis

In

Computer Science

COMSATS University Islamabad,

Lahore Campus - Pakistan

Spring, 2018

COMSATS University Islamabad,

Lahore Campus

Detection of Antipatterns from Service Oriented Architecture

A Thesis Presented to

COMSATS University Islamabad,

Lahore Campus

In partial fulfillment

of the requirements for the degree of

PhD (Computer Science)

By

Fatima Sabir

CIIT/SP14-PCS-003/LHR

Spring, 2018

Detection of Antipatterns from Service Oriented Architecture

A Post Graduate Thesis submitted to the Department of Computer Science as partial fulfillment of the requirements for the award of Degree of PhD in Computer Science.

Name	Registration Number
Fatima Sabir	CIIT/SP14-PCS-003/LHR

Supervisor

Dr. Ghulam Rasool

Associate Professor, Department of Computer Science

COMSATS Univeristy Islamabad,

Lahore Campus.

Co-Supervisor

Dr. Farooq Ahmad

Associate Professor, Department of Computer Science

COMSATS University Islamabad,

Lahore Campus.

Certificate of Approval

This is to certify that research work presented in this thesis, entitled "Detection of Antipatterns from Service Oriented Architecture" was conducted by Ms. Fatima Sabir, Registration No: CIIT/SP14-PCS-003/LHR, under the supervision Dr. Ghulam Rasool. No part of the thesis has been submitted anywhere else for any other degree. This thesis is submitted to the Department of Computer Science, COMSATS University Islamabad, Lahore Campus, in the partial fulfillment of the requirement for the degree of Doctor of Philosophy in the field of Computer Science.

Fatima Sabir Signature: _____

Examination Committee

_____ _____

External Examiner 1: Name
(Designation& Office Address)

External Examiner 2: Name
(Designation& Office Address)

_____ _____

Dr. Ghulam Rasool
Supervisor
Department of Computer Science,
COMSATS University Islamabad, Lahore Campus

HOD,
Department of Computer Science
COMSATS University Islamabad,
Lahore Campus

_____ _____

Chairman,
Department of Computer Science,
COMSATS University Islamabad

Dean,
Faculty of Information Sciences
and Technology
COMSATS University Islamabad

Author's Declaration

I Fatima Sabir, Registration # CIIT/SP14-PCS-003/LHR, hereby state that my Ph.D. thesis titled "Detection of Antipatterns from Service Oriented Architecture" is my own work and has not been submitted previously by me for taking any degree from this University i.e. COMSATS University Islamabad or anywhere else in the country/world.

At any time if my statement is found to be incorrect even after I graduate the University has the right to withdraw my PhD degree.

Date: March 4, 2019

Singature of the Student

Fatima Sabir
CIIT/SP14-PCS-003/LHR

Plagiarism Undertaking

1 solemnly declares that research work presented in the thesis titled "Detection of Antipatterns from Service Oriented Architecture" is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zerotolerance policy of HEC and COMSATS University Islamabd towards plagiarism. Therefore, I as an author of the above-titled thesis declare that no portion of my thesis has been plagiarized and any material used as areference is properly referred/cited.

I undertake if I am found guilty of any formal plagiarism in the above titled thesis even after award of PhD Degree, the University reserves the right to withdraw/revoke my PhD degree and that HEC and the university has the right to publish my name on the HEC/university website on which names of students are placed who submitted plagiarized thesis.

Date: March 4, 2019

Singature of the Student

Fatima Sabir
CIIT/SP14-PCS-003/LHR

Certificate

It is certified that Fatima Sabir (CIIT/SP14-PCS-003/LHR) has carried out all the work related to this thesis under my supervision at the Department of Computer Science, COMSATS University Islamabad, Lahore Campus, and the work fulfills the requirement for the award of the PhD degree.

Date: March 4, 2019

Supervisor:

_____

Dr. Ghulam Rasool

Associate Professor

Head of Department:

_____

Department of Computer Science

COMSATS University Islamabad,

Lahore Campus

Dedication

My Supervisor

&

My Husband

ACKNOWLEDGEMENT

I am thankful to my Creator Allah Subhana-Watala who guided me throughout the journey of my Ph.D research and the blessings of my beloved Holy Prophet peace be upon him that enlighten my path in this journey

I am especially thankful to my supervisors, Dr. Ghulam Rasool, for continuous support, guidance and encouragement throughout out my research. I am really honored for his guidance, patience, suggestions to improve my work. This journey will not be possible without his support. I am thankful for my co-supervisor for the feedback on tool and paper. I want to show my gratitude and unconditional support and external supervision by Dr. Yann-Gaël Guéhéneuc and Dr. Naouel Moha, for everything they did across the border. They guide me thorough out my research.

A special thanks message for Higher Education Commission of Pakistan for the IRSIP award granted to work under the supervision of Dr. Yann-Gaël Guéhéneuc at PTDIEJ lab and Dr. Naouel Moha for LATECE lab. I also want to thanks Dr. Francis Palma for his valuable feedback on tool and the suggestion for the improvement.

I am really thankful for the support of software engineering research group at COMSATS Lahore for testing and prototype guidance. A special thanks to Mr.Hassan Akhtar and our MS Students Mr.Umer Farooq and Ms. Maria for participating in research and verification of results. I would also like to thank for post graduate student of LATECE and PTDIEJ lab for conducting the case study.

My special thanks to Professor Dr. Syed Asad Hussain, Dean Faculty of Information Sciences and Technology, and Dr. Zulfiqr Habib, Chairman Department of Computer Science for their guidance and administrative support. I am really thankful to Dr Mudassar Naseer and Dr Waqas Anwar for helping me to follow the roadmap of doctorate degree.

In the end, I am really overwhelmed for the support of my husband for thorough out my degree especially when I was in Canada for research fellowship. He is the one who take care my parents too even I was not in Pakistan. In the last, I want to say thanks to my colleagues and my friends for their prayers and support.

Fatima Sabir

CIIT/ SP14-PCS-003/LHR

ABSTRACT

Detection of Antipatterns from Service Oriented Architecture

Web-services have become a governing technology for Service Oriented Architectures due to reusability of services and their dependence on other services. The evolution in service based systems demand frequent changes to provide quality of service to customers. It is realized by different authors that evolution in service based systems may degrade design and quality of service and may generate poor solutions known as antipatterns. The detection of antipatterns from web services is an important research realm and it is continuously getting attention of researchers. There are a number of techniques and tools presented for detection of antipatterns from object oriented software applications but only few approaches are presented for detection of antipatterns from SOA. The state of the art antipattern detection approaches presented for detection of antipatterns from SOA are not flexible.

We present a flexible approach supplemented with a tool support named as SWAD(Specification of Web service Antipatterns Detection) to detect antipatterns from different SOAP based applications. Service-based systems, in particular, RESTful APIs, need to meet both service consumers' and providers' requirements. Similar to other software systems, RESTful APIs face continuous maintenance and evolution. Applying poor design principles called antipatterns, may hinder the maintenance and evolution of RESTful APIs, as compared to the good design principles, i.e., design patterns that facilitate maintenance and evolution. Antipatterns may also affect the usability of RESTful APIs. Major market players like Facebook and YouTube are already using REST architecture and their APIs are frequently evolving to meet the end users' requirements. Although, a number of antipatterns are defined in the literature and researchers performed their automatic detection but the evolution of RESTful APIs did not receive much attention. There is a need to track the evolution of antipatterns in the RESTful APIs that could assist service providers publishing well-designed and easy to consume RESTful APIs for their clients. In this dissertation, we present the correction of eight REST antipatterns in RESTful APIs with a tool support called SOCAR (Service Oriented Correction of Antipatterns in REST) after analyzing their evolution history for two years. Our correction heuristics are validated by practitioners with an average precision of 100% and an average recall of 94%. Moreover, we propose a methodology for the correction of linguistic

antipatterns with a tool support COLAR(Correction of Linguistic Antipatterns for RESTAPIs).

Table of Contents

Chapter 1 Introduction..... 2

1.1 Research Context..... 2

1.2 Problem Statement and Thesis..... 6

1.3 Specification, detection, evolution and correction of antipatterns..... 8

1.4 Contributions..... 8

1.5 Organization of the Dissertation..... 9

Chapter 2 Building Blocks of SOA..... 11

2.1 Services..... 12

2.2 Roles of Services..... 12

2.3 Effective use of services..... 13

2.4 Composition of Services..... 14

2.5 Implementation approaches for SOA..... 14

2.5.1 WSDL..... 15

2.5.2 SOAP..... 16

2.5.3 OPC UA..... 17

2.5.4 REST full Services..... 17

2.5.6Apache Thrift..... 18

2.5.7 Message Oriented Middleware..... 18

2.6 SOA technologies and Quality of Service issues..... 19

2.7 Antipatterns as a quality indicator for SOA technologies..... 22

2.8 Discussion..... 23

Chapter 3 Systematic Literature Review..... 25

3.1 Introduction..... 26

3.2 Research Methodology for SLR..... 30

3.3 Classifications of the State-of-the-art Techniques Employed in the Detection of Smells..... 43

3.3.1 Static Source Code Analysis 44
3.3.2 Dynamic Source Code Analysis 53
3.4 Evolution of State-of-the-art Approaches 54
3.4.1 Source Code Metrics 54
3.4.2 Mining the Source Code using SVN or CVS 55
3.4.3 Domain Specific Language 56
3.4.4 Genetic Algorithm 57
3.4.5 Parallel Evolutionary Algorithm (PE-A) 58
3.5 Bad Smells that are studied for a specific Paradigm 59
3.5.1 Smells Reported in OO 59
3.5.2 Smells Reported in SO Systems 65
3.6 Correlation between Smells across the Paradigms 68
3.7 Trends in Research on Smells from January 2000 to December 2017 72
3.8 DISCUSSION AND OPEN ISSUES 74
3.9 Summary of Literature Review 77
Chapter 4 Specification of Web Services Antipatterns Detection 80
4.1 Introduction 81
4.2 State of the Art for Web services 83
4.3 SWAD Approach 88
4.3.1 Specification of SOAP Antipatterns 88
4.3.2 Detection Approach for SOAP Antipatterns 91
4.4 Experimental Results 96
4.4.1 Results of Antipattern Detection using Code –first Approach 99
4.4.2Results of Antipattern detectionat interface level using Contract first Approach 102
4.4.3 Comparison of Results 108
4.4.4 Comparison of Results with P.E Algorithm 110

4.5Conclusion 113

Chapter 5 Service Oriented Correction of Antipatterns for RESUTful APIs 114

5.1 Introduction 115

5.2 Related work 118

5.3 Study Design 124

5.3.1 Data Extraction and Analysis 126

5.3.2 Correction of Antipatterns using Evolution History 131

5.4 SOCA-R(SERVICE ORIENTED CORRECTION OF ANTIPATTERNS for REST APIs) 136

5.5 Analysis of Results 140

5.5.1 RQ1: When antipatterns are introduced? 140

5.5.2 RQ2: How antipatterns are evolved? 146

5.5.3 How antipatterns are removed? 148

5.6 Threats to Validity 149

5.6.1Construct Validity: 150

5.6.2 Internal Validity: 152

5.5.3 External Validity 152

5.6 Conclusion 153

Chapter 6 Correction of Linguistic Antipatterns for RESTful API 155

6.1 Introduction 156

6.2 Related Work 157

6.2.1 Analysis for Web services 157

6.2.2Analyses for OOSE. 159

6.3 Correction of Linguistic Antipatterns in REST APIs. 160

6.3.1 Definition Analysis for RESTful Linguistic Patterns and Antipatterns 161

6.3.2 Implementation of Correction Algorithms 167

6.4Analysis of Results 167

6.4 Discussion on Results 170
6.5 Validity:..... 171
Threats to Validity 173
6.6 Conclusion..... 174
Chapter 7 Conclusion 175
7.1 Conclusion 176
7.2 Implications of Research..... 177
Chapter 8 References..... 179
Appendices..... 201
Appendix I: Primary studies used for SLR 202
Appendix II: Evaluation of REST API 206
Appendix III: Correction Algorithm for REST Antipatterns 212

List of Tables

Table 2.1 Quality Attributes for SOAP and REST 19
Table 3.1: Overview of Existing Reviews 29
Table 3.2: Research Questions 32
Table 3.3: Number of Studies Found in Selected Digital Libraries after General Term Search. 33
Table 3.4: Data Extraction Sheet. 37
Table 3.5: Techniques for Continuous Studies. 39
Table 3.6: Frequency of Smells Terms in Studies 40
Table 3.7: Studies Used Behavioral Source Code Analysis 45
Table 3.8: Empirical Source Code Analysis Techniques. 46
Table 3.9: Algorithm-based Source Code Analysis. 49
Table 3.10: Methodological Source Code Analysis. 51
Table 3.11: Linguistic Source Code Analysis. 52
Table 3.12: Dynamic Source Code Analysis Techniques 53
Table 3.13: Smells Reported in the Literature from the OO Paradigm 63
Table 3.14: Smells that are Reported Repeatedly in the Services Literature 66
Table 3.15: Smells Reported for the First Time in the Services Literature. 67
Table 3.16: Smells Reported for the First Time in the Services Literature. 67
Table 3.17: Smells Evolved in OO and SO. 69
Table 3.18: Source Code Metrics used for the Detection of Services Smells. 70
Table 3.19: Trends in Research from the year 2000 to 2017. 73
Table 4.1 :Summarized Information about SOA Antipattern Detection Techniques 85
Table 4.2 : Mapping of SOAP interface with SOAML Model 95
Table 4.3 : Statistics of Examined Systems 97
Table 4.4 Results for Finance related Web-services 99
Table 4.5 Results for Weather-related Web-services 101
Table 4.6 Results for Finance related Web-services 103
Table 4.7 Results for Weather-related Web-services 107
Table 4.8 Description of Detection Tools 108
Table 4.9 Comparison of Results for Weather Related Services 108
Table 4.10 Comparison of Results for Finance Related Services 109
Table 4.11 Comparison of Results Generated by SWAD 112
Table 5.1 List of RESTful APIs under Analysis 128
Table 5.2 Operations Performed in Evolution 130
Table 5.3 Refactoring Operations Performed for each Antipattern 139
Table 5.4 Evolution History of REST APIs 144
Table 5.5Antipattern Detection for Stack Exchange and Facebook Version History 145
Table 5.6 Correction of Antipatterns by SOCA-R 149
Table 5.7 Accuracy of SOCA-R(Service Oriented Correction for Antipatterns for REST API 151
Table 6.1 Online documentation of Services Analysed 168
Table 6.2 Results of correction of Linguistic Antipatterns 169
Table 6.3 Relative Accuracy Measures of COLAR Tool 173

List of Figures

Figure 2.1 Interaction of Services in SOA [118] 12
Figure 2.2 Interaction among Service Components[118] 13
Figure 2.3 Implementation view of SOA [118] 15
Figure 2.4 Communication between WSDL Document Components [118] 16
Figure 2.5 SOA Technology used by Industry [228] 20
Figure 2.6 SOAP and REST Architectural Difference [228] 22
Figure 3.1: Software Architecture with Various Domains 26
Figure 3.2: Steps Followed for Systematic Literature Review. 31
Figure 3.3: Study Selection Criteria 36
Figure 3.4: Distribution of Studies w.r.t. the Data Analysis Techniques 40
Figure 3.5: Data Source Used for the State-of-the-Art Research on Smells 41
Figure 3.6: Year Wise Distribution of Studies. 42
Figure 3.7: Year Wise Distribution of Studies w.r.t. Different Paradigms. 43
Figure 3.8: Distribution of Primary Studies based on Source code analysis 45
Figure 4.1 Antipatterns Detection Approach For SOAP Services 96
Figure 4.2 Interface of Prototype Tool 96
Figure 5.1 Research Methodology 129
Figure 5.2 Migration Traces of Alchemy RESTful API 133
Figure 5.3 Migration Traces of Alchemy API 134
Figure 5.4 Correction Heuristic of Mime Type Antipatterns 135
Figure 5.5 Correction Heuristic of Forgetting Hypermedia Antipattern 135
Figure 5.6 Antipatterns Evolution for Alchemy API 146
Figure 5.7 Antipatterns Evolution for Bitly API 147
Figure 5.8 Antipatterns Evolution for YouTube API 148
Figure 6.1 Correction of Linguistic Antipatterns 161
Figure 6.2 Correction Heuristics for CRUDY Antipatterns 162
Figure 6.3 Correction Heuristics for Hierarchical Nodes Antipatterns 163
Figure 6.4Correction Heuristics for Amorphous Antipattern 165
Figure 6.5 Correction Heuristics for Pluralised Antipatterns 166
Figure 6.6 Traces of Antipattern Correction for Twitter 171
Figure 6.7.Traces of antipattern correction for YouTube 171

LIST OF ABBREVIATIONS

JSON	JavaScript Object Notation
OMG	Object Management Group
OOP	Object-Oriented Programming
QoD	Quality of Design
QoS	Quality of Service
REST	REpresentational State Transfer
ROI	Return On Investment
SBSs	Service-based Systems
SCA	Service Component Architecture
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SODA	Service Oriented Detection for Antipatterns
SOFA	Service Oriented Framework for Antipatterns
UDDI	Universal Description, Discovery, and Integration
URIs	Uniform Resource Identifiers
WSDL	Web Service Description Language
XML	eXtensible Markup Language
SOCA-R	Service Oriented Correction of Antipatterns for REST
COLAR	Correction of Linguistic Antipatterns for REST
OO	Object Oriented
SO	Service Oriented
SBS	Service Base System

List of Publications

1. Sabir, F., Rasool, G. and Yousaf, M. (2017). A Lightweight Approach for Specification and Detection of SOAP Anti-Patterns. *International Journal of Advanced Computer Science and Applications*, *8*(5): 455-467.
2. Sabir, F., Rasool, G., Palma, F., Moha, N., and Guéhéneuc, Y. (2019). A Systematic Literature Review on the Detection of Smells and their Evolution in Object-Oriented and Service-Oriented Systems. *Journal of Software practice and experience*, *49*(1), pp.3-39.
3. Sabir, F., Rasool, G., Palma, F., Moha, N., Guéhéneuc, Y., and Akhtar, H. (2018). Correction of Antipatterns in RESTfull APIs using evolution history. *Submitted to possible publication to Empirical Software Engineering (ESE).*
4. Sabir, F., Rasool, G., Farooq, U., Francis, P., Moha, M. Guéhéneuc, Y. (2018). Correction of Linguistic Antipatterns in RESTfull APIs, *submitted for possible publication to International Journal of Cooperative Information Systems (IJCIS).*

Chapter 1
Introduction

Chapter 1 Introduction

1.1 Research Context

Service-Oriented Architecture a building block for Service Oriented Programming

Service Oriented Architecture (SOA) is an architectural style for building solutions in the form of services for enterprise applications. SOA is specifically used for the construction of independent business intelligence services which can be combined into meaningful, higher business processes and solutions within the framework of an enterprise. The real essence of SOA is the use of integrated reusable, agile and flexible business processes. Each business process performs separate tasks in the organization and the world of e-business provides information to automate different business processes across the organization; which in short elevates the need of standardized protocol and web service composition languages which can be understood across the industry. The most modern trend in business process management is the use of information technology. Organizations use different software and web services to run these business processes.

As per the definition of IEEE systems, software engineering is defined as "*the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software*". The major key principal of software engineering is to address poor quality of software, keep track of project deadlines and ensure that software is built systematically as per the specification and requirements of client. One of the earlier software development is the procedural development which follows the sequential execution of the program. However, there are several drawbacks behind the functional programming practices: (1) it is difficult to implement real world concept programmatically; (2) systems are more complex as code grows; (3) Data is centralized and hence not secured i.e. publically available. These problems are addressed in the form of Object –Oriented Programming which promotes reusability and avoids exposing the inner detail as per the user requirement and improves the software performance. As an organization grows and internet came into existence, organizations want to grow their businesses across the internet as well as transactions done over the web. It introduces the new software development dimension known as

Service Oriented Programming which promotes the remote access and platform independent functional units also known as services. Service oriented programming introduces an additional layer of software abstraction named as service layer.

Internet is used as a bridge which helps SOA to provide a large number of simple services that can be integrated with the help of different protocols to develop new and complex services. Service oriented programming helps to integrate services but these services use complex architecture which creates problem for clients who use these services due to the many reasons: 1) Services are mostly deployed and developed independently. 2) Encapsulation is the key principle which prevents external users to access the internal structure of the services. 3) Interfaces are available for the users which are the major attraction for software industry. Recent standards of WSDL (Web Service Description language) and WIDL (Web Interface Definition Language) are emerging and play the same role as IDL (Interface Description Language) in different component technologies.

Integration of services using Service Oriented Programming (SOP) helps the services to integrate components easily. Integration of components with services can be performed using a specific service handler which uses specific protocols and converts request from one communication protocols into another one. Services are not platform dependent: The suitable platform can be chosen based on the application type and complexity.

Services are grouped together based on the needs of the users in service based systems and use SOA as an architecture style[2]. Service based system uses services as its building blocks [118][3]. SOA is gaining popularity and strongly supported by the major software vendors like SAP, Sun, IBM which provide different development tools. SBS (Service Based Systems) is designed and developed with the help of SOA design patterns, principles [118] and different technologies like SCA (Service Component Architecture) [5], SOAP[4] and REST (REpresentational State Transfer) [6]. In this dissertation, we mainly focus on REST and SOAP based web services which are widely used by financial, educational and telecom industries.

Maintenance of Web services and their evolution

Maintenance and evolution of the service base systems is a new term as compared to software system evolution, as most of the web services have been introduced after 1999. Tracking the evolution of service oriented systems is quite complex due to the distributed nature of services, whose multiple parts are available on different servers which also help the multinational companies to monitor their business globally. However, as the design of these services is expressed in terms of the interface specification, there is a need to identify the changes which may impact these interface specifications. Frequently, service implementation is totally unknown to the clients who use these services, while the service providers have no idea about the users of these services. Service providers may change their service at any time; however, users of these services must be aware of these changes. The changes in Service Oriented (SO) systems suffered by the problems due to the multiple service consumers and providers, iteration at irregular bases due to the change in business needs and migration from one SOA technology to another technology.

Quality of Service (QoS) is suffered due to the rapid change in service interface to meet current user requirements. There is a need to adopt controlled mechanism in message technology (REST, SOAP) between client and servers which maintain QoS. However, there are wide ranges of tools used in OOSE (Object Oriented Software engineering) to support analysis and maintenance of legacy systems. But, specialized methods and tools which may assist developers to keep track of services for the maintenance and evolution are still in its infancy. Migration of legacy systems towards SOA is still a big challenge as there is no automated process which keeps track of each phase and assists software maintenance team during migration.

Antipatterns to track Software Quality

Antipatterns are normally described the commonly generated solution of a problem which may generate negative results [8] as compared to the design patterns which improve the software quality and decrease maintenance cost. They also degrade the system performance and may increase the maintenance time. There is a need of real world experience and common vocabulary to identify the Antipatterns [40]. These Antipatterns are another form of code smells introduced by Brown et al. [40]. There are number of studies which assess the software quality by investigating the impact of

design patterns and antipatterns [112][10][77]. Different researchers investigated the effect of antipatterns on system maintenance [10][77] as well as their effect on program understandability which provides a reason for antipatterns evolution from one version to another.

It is also very important to evaluate the performance of the classes which have antipatterns as compared to the classes which are free from antipatterns. ABS (Antipatterns identification using B-Splines) is the technique which focuses towards the detection of antipatterns in broad line classes [78]. OOSE (Object Oriented Software Engineering) antipatterns are also presented in [87, 143] which have little variation of other design principles. Antipatterns detection with model queries and formal concept analysis are other common approaches as well [142,139].

The major focus of the previously published research was antipatterns and code smells detection in OOS (Object Oriented Systems), while only limited work is done on Service Oriented Architecture (SOA) Antipatterns.

Antipatterns detection from Service based systems was introduced by Palma with a framework support called SOFA (Service Oriented Framework for Antipatterns) [66] [68]. The effect of services which are detected as antipatterns needs more maintenance as compared to those services which are not detected as antipatterns [68]. SOA antipatterns may reduce maintainability and reusability in Service based systems [67]. SOMAD (Service Oriented Mining for Antipatterns Detection) improves the previously mentioned SODA(Service Oriented Detection for Antipatterns) approach and improves its precision by introducing some new suites of dedicated metrics [67]. This approach used mining rules from the field of data mining and detect strong association among sequence of services/ method call and then filter them using the metric suite. Antipatterns also evolve not only in the OOSE but also in web services which need serious consideration.

From the aforementioned studies, it is clear that keeping track of antipatterns detection along with evolution is necessary to maintain the services since only few studies reported antipatterns detection for SOAP and REST web services, and the evolution of these antipatterns varies for each SOA technology like SOAP and REST. Detection of antipatterns from services using different SOA technologies like SOAP and REST is challenging because architectural style of SOA technologies is different.

1.2 Problem Statement and Thesis

Antipatterns detection affects the service availability and maintenance. Pressman[144] focused on the detection and correction of design problems which helps developers to reduce the future maintenance cost and efforts.

There are different approaches which report the detection of antipatterns from services [66][68][56][57][5][47] but detection of the antipatterns depends upon the technology being used by the service providers. Evolution of antipatterns is reported for SOAP based services [121] but not for REST based services yet. Moreover, correction of these antipatterns is not taking much attention and only subjective assessment is reported without the tool support which prevents service system providers like Facebook, YouTube or Google to keep track of their antipatterns evolution and automatically detect (and correct) those antipatterns.

There is need to provide antipatterns detection technique for SOAP web services free from any dependency of platform as reported in literature and correction of antipatterns for RESTful APIs that still not get much attention along with evolution history .

Problem 1. No generic approach is available for the detection of antipatterns using variable threshold adaptation from multiple technologies:

There is not a single publically available tool for the detection of antipatterns based on automated threshold adaptation both from industry standard code-first as well as contract first approach used for SOAP web services. Despite the commonalities of two approaches used by SOAP web services; there is a need of generic approach which can be used by both techniques.

Problem 2. Evolution of Services:

Services are growing continuously due to the rapid change in user requirements. There is a need to assess the evolution of antipatterns along the evolution of services to prepare a catalog that helps the industry to maintain services. The present information is not documented and is available in textual format without clear facts and figures.

Problem 3. No single approach is available that reports the correction of antipatterns for REST web services:

Despite the availability of multiple services for OOSE which report the correction of antipatterns for OOSE, there is not a single approach which deals with the antipatterns correction for web services for different SBSs systems that operation the Internet-based dynamic environment.

Problem 4. No single approach is available that reports the correction of linguistic antipatterns for REST web services:

There are number of approaches available that report the detection and correction of linguistic antipatterns for OOSE. However, the correction of linguistic antipatterns for RESTful web services is still not available that will help the developers to maintain the URI of RESTful APIs.

In the existing literature, above four problems are not addressed which provides a baseline for this thesis statement as following:

Thesis: "Specification, detection, evolution and correction of antipatterns for web services ".

Above thesis statement is further supported by research questions as given below:

RQ1 **–** *What type of customizable approach that can be used by both academia and industry for the detection of antipatterns for SOAP web services?*

RQ1 is answered after testing our technique generically for SOAP based web services. As SOAP based web services used either code-first or contract-first approach, so our framework is applicable for any technique used for managing SOAP based services.

Solution to Problem 1: A simple lightweight approach is reported that can be used by any industry standards for creating the SOAP based services.

RQ2– *How antipatterns are evolved for REST APIs?*

Research question 2 is answered after covering evolution history for REST API over the past three years. To conclude with RQ2, we investigate the following:

Keeping track of antipatterns detection traces along with their version history over the past three years to know the reason of antipatterns evolution.

RQ3 – *How antipatterns for REST services are corrected?*

Solution to Problem 2,3 and 4 : A generic approach that helps to correct the REST antipatterns dynamically. We apply correction algorithm with the help of trace history. We also implemented the correction algorithms for linguistic antipatterns for REST APIs to improve the REST APIs URIs.

1.3 Specification, detection, evolution and correction of antipatterns

Step 1: To answer RQ1, which deals with the detection of antipatterns from SOAP web services, we firstly analyze the different approaches being used by the industryfor designing SOAP web services and then propose a generic approach that can detect antipatterns from web services based on the industry standards. This approach is tested on the publically available datasets. SQL (Structured Query Language) and regular expressions are used with dynamic threshold adaptation set by the users. The SWAD (Specification of Web service Antipatterns Detection) tool also usesWordNet [11] and Stanford CoreNLP [13] for checking the similarity of web operations used by SOAP web services.

Step 2: We use SODA-R proposed by the Palma et al. [56] to check the evolution history of web services over 2015-2017 to instigate the effect of antipatterns on major REST API service providers. Finally, we collect the real time traces of REST API detection to investigate the major changes adopted by the famous REST API providers to mine the antipatterns as well as design patterns along the version history.

1.4 Contributions

This dissertation provides a novel approach for the specification, detection, evolution and correction of SOA antipatterns. The main contributions are as follows:

- A lightweight approach easily customizable for different SOA technologies for the detection of antipatterns. We implemented our approach for SOAP web

services and evaluated our approach for two industry standards. **(Addresses the Problem 1)**

- An empirical evidence after mining the trace history of REST API header, body, status code and other parameters over the past two years. The changes in request, response are collected from real time traces (**solving Problem 2**);
- Using the real time definition of corrected parameters of request and response used by the REST API providers to prepare a tool which automatically corrects the antipatterns (**solving Problem 3**);
- An extensive validation of the SWAD and SOCA-R tool by the industry and academia to calculate precision and recall metrics. (**solving Problem 2 and 3**);
- SOFA framework is extended for the correction of REST linguistic antipatterns (**solving Problem 4**)

1.5 Organization of the Dissertation

The remainder of this dissertation provides the following content:

Chapter 2: Background provides a background on Service Oriented Architecture

especially two important technologies used by the SOA. It discusses the different technologies lying under this umbrella and how these technologies are used by major SOA service providers with the help of discussion, commonalities and comparisons among those technologies. We also discuss the feature type of design issues and how these technologies evolve with poor design practices called as antipatterns.

Chapter 3: Literature Review performs a systematic literature review on the existing methodologies and discusses the various issues or source code problems evolving from an OOSE to SOA. Finally, we report the existing gaps in the literature that provides the base for this thesis.

Chapter 4: Specification and Detection of SOAP Antipatterns presents the lightweight approach which covers two industry standards used to make SOAP web services. The specifications of SOAP antipatterns are reported.

Chapter 5: Correction and detection of REST Service Antipatterns using Evolution History presents extensive mining of trace history of REST antipatterns evolving across various versions of REST APIs or from one version to other versions. Mining of evolution history will help to investigate the changes committed by each

REST API provider and prepare a catalog which helps to know REST service provider's industry trends. Results of evolution are discussed and findings are reported.

Chapter 6: Correction of REST Linguistic Antipatterns reports the technique used for the correction of linguistic antipatterns as well as impact of correction being further evaluated by the industry.

Chapter 7: Conclusion presents the conclusion of this dissertation and outlines some directions for future research.

Appendix A: Primary studies use for systematic literature review

presents the list of primary studies used in SLR for problem overview.

Appendix B: Validation of REST Antipatterns

Presents the questionnaire used for the validation of REST antipatterns.

Chapter 2 Building Blocks of SOA

2.1 Services

A service is a software component used by a network which helps to fulfill the request of client users by authentication of service providers. Services can be used individually and helpful like *Yahoo weather forecasting* or can be integrated with other services to provide more complete information: i.e. use of weather forecasting for the prediction of a live cricket match. Integration of services is possible with the help of exchanging messages between multiple services and collecting response of these messages. Internal and external communications via services are possible due to the hierarchy followed by sent and receive messages. These messages affect the results of the operations used by services defined as "service choreography". Services are either independent or dependent like a currency conversion service might depend on an online service since currency exchange rates vary dynamically.

Figure 2.1 explains the way services interact: A client which may or may not be another service can request the registry to search for a particular need and the registry replies with the list of suitable services. The client can select any service as per one's requirement with the help of mutually agreed protocols. Service responds either with the desired results or with the default message.

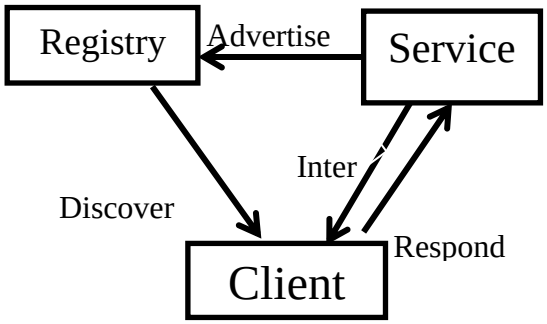


Figure 2.1 Interaction of Services in SOA [118]

2.2 Roles of Services

There are different ways to describe the services, however, some different terms are associated with the services especially for web services which are discussed below [118]:

Service provider is a software entity which helps to apply service specification in multiple ways as either internal, shared or external service provider.

Service requestor is software entity which requests from service provider in the form of client or it can be an external service as well.

Service locator is also a service provider which provides specific services and works as a registry for communication between service provider interfaces and service location.

Service broker is another form of service provider which can transfer request to one or more service providers.

The roles of aforementioned indicators are also described in Figure 2.2

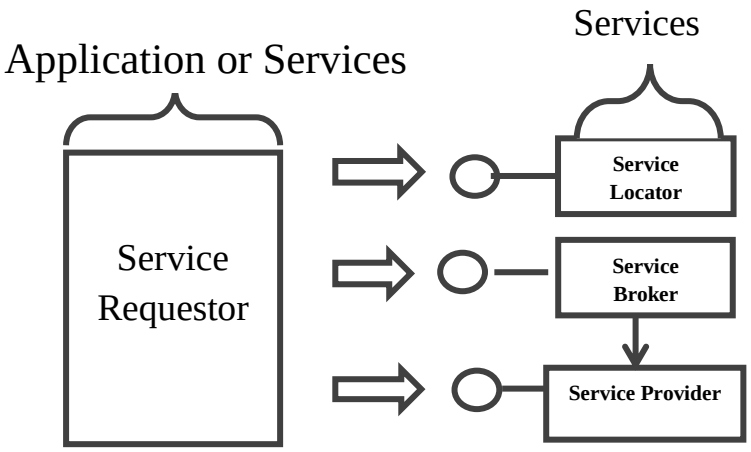


Figure 2.2 Interaction among Service Components[118]

2.3 Effective use of services

The description of services, and the context of their use imposed a series of constraints. Furthermore, efficient use of services suggests a few high-level best practices. Here are some key characteristics for effective use of services [3]:

Coarse-grained — Operations on services are frequently implemented to encompass more functionality and operate on larger datasets, compared with component-interface design.

Interface-based design — The implementation of services defines interfaces separately. The benefits of interface-based design are that multiple services can implement a common interface and a service can implement multiple interfaces.

Discoverable — Services need to be found at both design time and run time, not only by unique identity but also by interface identity and by service kind.

Single instance — Unlike component-based development, which instantiates components as needed, each service is a single, always running instance that a number of clients communicate with.

Loosely coupled — Services are connected to other services and clients using standard, dependency-reducing, decoupled message-based methods such as XML document exchanges.

2.4 Composition of Services

Service composition is the design element of SOA which promotes the design of services in a way that can be further reused in the multiple solutions. Service composition will be at its maximum ease if the services are independent of their size and composition of service must fulfill the service loose coupling principle.

The softwares are developed using existing components promoting the concept of composition. This is the key principles of object oriented software engineering where resultant product is composed of relevant interlinked objects which provide solution of multiple problems. The same concept is applied in SOA where focus is to build services that can be reused by using multiple solutions fulfilling the criteria of agility. Service composition principles prepare the services in a way that helps to implement any change in future design of service composition.

2.5 Implementation approaches for SOA

Service Oriented Architecture is applied in an application with the help of web services. Implementation of SOA is possible with the use of standard internet protocols which can be used without restriction of any programming language and platform [118] [3]. These services can be used as a new application or wrapper of existing legacy applications which continuously evolve with respect to time. Service system designer or programmers mostly build services by using web service standards. These standards also follow the principal properties of SOA mentioned in first section of this chapter. Architecture can work independent of any specific technology and therefore can be implemented with the help of technologies as described in Figure 2.3.

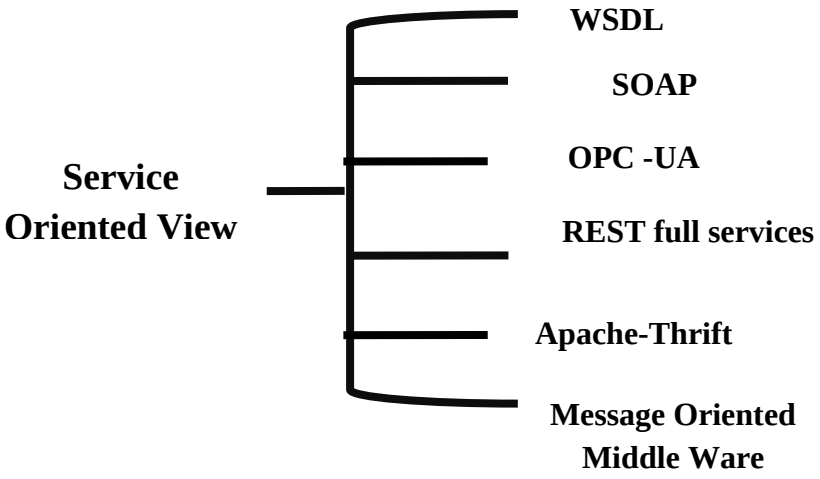


Figure 2.3 Implementation view of SOA [118]

2.5.1 WSDL

Web Service Description Language (WSDL) is an XML (Extensible Markup Language) which helps service providers to design the functionality of their services as a combined set of operations with inputs and outputs. These set of operations also help defining the inputs and outputs which helps the consumers to benefit from the operations. Every input and output have different forms of data types written in schema definition (XSD) of XML. The service providers used specific techniques to make their WSDL document publicly available in service registries known as Universal Description, Discovery and Integration (UDDI). UDDI is used to represent meta-data about web services. UDDI registries are further classified as a semantic based registry which helps customers to search for a specific service based on their functionality. These type of registries use semantic annotation language like ontology web language. Another type of UDDI is information retrieval which can broadcast service without any specification and helps WSDL document to collect different terms. These terms helped todescribe the association among services and match specific services with the queries of service consumers. WSDL document has several version and currently used version is 2.0 because version 1.1 is no more accepted by the W3C. WSDL 2.0 is the alternative form of WSDL 1.1.

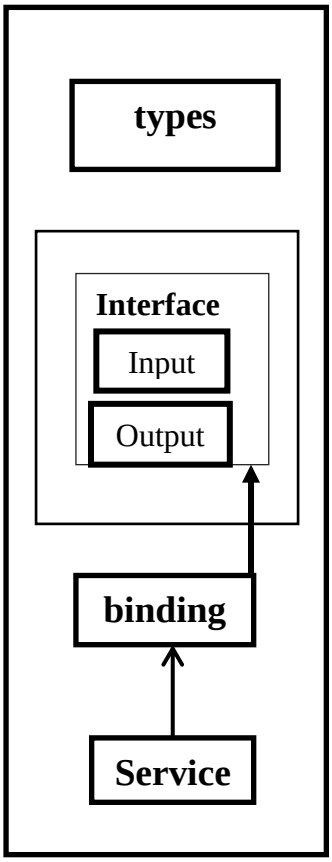


Figure 2.4 Communication between WSDL Document Components [118]

A WSDL use services as a combination of endpoints used in networks or in port. WSDL separates the definition of message from its network deployment or binding used to format the data. WSDL defines ***messages*** for data definition and ***port-types*** as collection of operations. The term ***port*** is used to bind the network address and ***service*** as a collection of endpoint.

2.5.2 SOAP

The architecture used by RPC is based on static typing of IDL

(Interface Description Language) which focused on tight coupling. The threshold level of coupling is due to IDL. The backbone of web services is to use low coupling due to the use of open standards. Web services mostly use XML document which encapsulates the distribution of shared objects among different interfaces with the help of loose coupling. SOAP (Simple Object Access Protocol) is the technology that

also relies loose coupling and shows communication along messages using XML document.

The WS-* is an extended version of SOAP based webservice which supports communication with improved reliability, security and communication among various business processes. The coupling between SOAP based web services is loose due to the support of XML and communication between messages and interfaces is in WSDL format. As per the definition of WSDL 2.0, an interface is defined as a set of operations and binding of operation is supported by HTTPS service which implements the interface at the service endpoints. XSD is used to define the message format for communication between interfaces with the help of input and output operations for incoming and outgoing communication. An endpoint defines the accessible address of URL (Uniform Resource Locator) which helps to consume the service with its concrete binding. The formation of SOAP services must follow the schema used for WSDL message type and encapsulates data in XML compatible text.

2.5.3 OPC UA

It's a communication protocol used by industry for automation and helps to establish communication between machines. OPC UA is mostly used by industrial equipment and systems for data collection. The OPC UA is available under the GPL 2.0 license. It can be used by any operating system or utilized with the help of any programming language which can shift it under the umbrella of SOA. The best feature of OPC UA is the support of redundancy, bidirectional communication even in the case of interruption between client and server and buffering of data along with its acknowledgement. The protocol supported by OPC UA is binary i.e. **http://Server** and **opc.tcp://Server** .

2.5.4 REST full Services

REpresentational State Transfer (REST) was first time introduced by Fielder in the year 2000. REST originally introduced an architectural style which was specifically used for large scale distributed systems. Its major component is the use of abstract entity with high scalability. REST has these basic properties:

Identification of resource through URI: Resource and service finding resource identifiers provide space globally available for resource and service discovery.

Uniform interface: Four different type of operations manipulate the resources using PUT, GET, DELETE and POST operations. Messages can be encrypted in multiple formats (PDF, JPEG, XML, JSON). Complete metadata information is available for multiple uses like transmission issues detection and resource availability.

Statelessness: Messages are delivered in request are self-contained.

RESTful Web services are following simple architecture followed by IETF/W3C . Their services and the users of these services are available in all major programming languages. Furthermore, web resources are discovered due to the URI and hyperlinks without the registration to a centralized repository.

2.5.6Apache Thrift

The Apache thrift is framework used for scalable multi language service development with the help of software layers which enable the code generation to build services. These services are supported by multi languages like Javascript, C++, Python, Ruby, Perl, Cocoa, Erlang, OCaml, Delphi and many more. Apache Thrift allows user to define multiple data types and different service interfaces in a definition file. The definition file is used as input and compiler automatically generates the code. The code is sent to build RPC (Remote Procedural Clients) that communicates logically with all types of programming languages. The user can now directly communicate with business layer instead of overlapping with multiple code properties or class files to invoke objects and call methods. Thrift uses multiple procedures as a default like binary format or in JSON and transfers from one layer to another using TCP or HTTP. The best example of the framework used by thrift is Twitter Finagle[217]. The Finagle is enhanced sing multiplexing; a technique used for transferring multiple signal across one link. A protocol used by Twitter is Mux lies between TCP and Thrift [217]. The parallel thrift connection runs between client and service with the help of single connection [217].

2.5.7 Message Oriented Middleware

Message-Oriented Middleware (**MOM**) is a communication layer which provides support for sending and receiving to/from distributed systems using software or hardware infrastructure. The applications using MOM can be distributed across multiple platforms and decrease the complexity level for applications using multiple

operating systems and communicate with each other using multiple protocols. APIs available at multiple platforms and networks mostly use MOM. A major reason of using MOM is its ability to transfer, save and route messages during sender and receiver conversations. Another benefit is that, by adding the administrative interface, one can check and adjust the performance. This will help the client to not care about any issue regarding application state management and enjoy its communication via sending and receiving message with server.

2.6 SOA technologies and Quality of Service issues

SOA uses multiple technologies and is covered by different types of protocols for sending and receiving messages between client and server. We mainly focused on SOAP and REST web services due to its usage by the companies. Table 2.1 highlights the key points for a comparison between SOAP and REST quality of service features collected from [118][3][5][6].

Table 2.1 Quality Attributes for SOAP and REST

Sr. No	Quality Attributes	REST	SOAP
1	Architecture Style	Client-Server (Architecture Driven)	Client –Server(Protocol)
2	Protocols	HTTP	HTTP,SMT,TCP
3	Processing	Not required	Extensive Processing due to WSDL
4	Security	SSL at transport layer	HTTPS
5	Message Content Format	XMl,JSON object Small Format	RPC, WSDL Long Format
6	Response Format	RSS ,CSV,JSON	XML
7	Bandwidth Requirement	Low	High
8	Cache Control	Cached	Not Cached
9	Web Service Message	Just retry in case of message failure	Retry logic already build

10	Code on Demand	Available	Not Available
11	Data Payload	Uses HTTP,JSON	Uses XML
12	Usability	Interaction with REST API is User independent	User must know about API for communication
13	Data Availability	Availability of data as resource (‘noun’)	Availability of data as Service (‘getuser’,’Payinvoice’)
14	Best Use for	Entertainment ,social industry	Payment, Financial, Transaction
15	Famous Users	Facebook, Twitter, YouTube	PayPall, Salesforce, Clickatell

According to the statistics collected from survey by InfoQ in 2011 [228], REST is widely adopted architecture style by industry followed by SOAP. Figure 2.5 highlights some important statistics of SOA technology used by the industry as well as format of data adopted by video APIs.

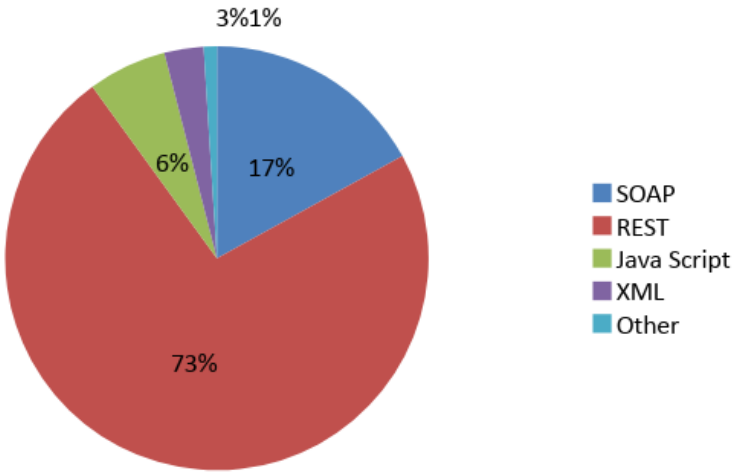


Figure 2.5 SOA Technology used by Industry [228]

Figure 2.5 highlights the statistics collected from programableweb.com regarding maximum SOA technologies used: 90 percent belongs to SOAP and REST and the other technologies portion is still 10%. The widely adopted SOA technology and it’s industry use motivated us to work in SOA technologies and focus specifically for the SOAP and REST web services.

REST and SOAP services can also be differentiating with not only the architectural style but also the way each service used different components and provide end user operation This can also be differentiate with the help of diagram 2.6

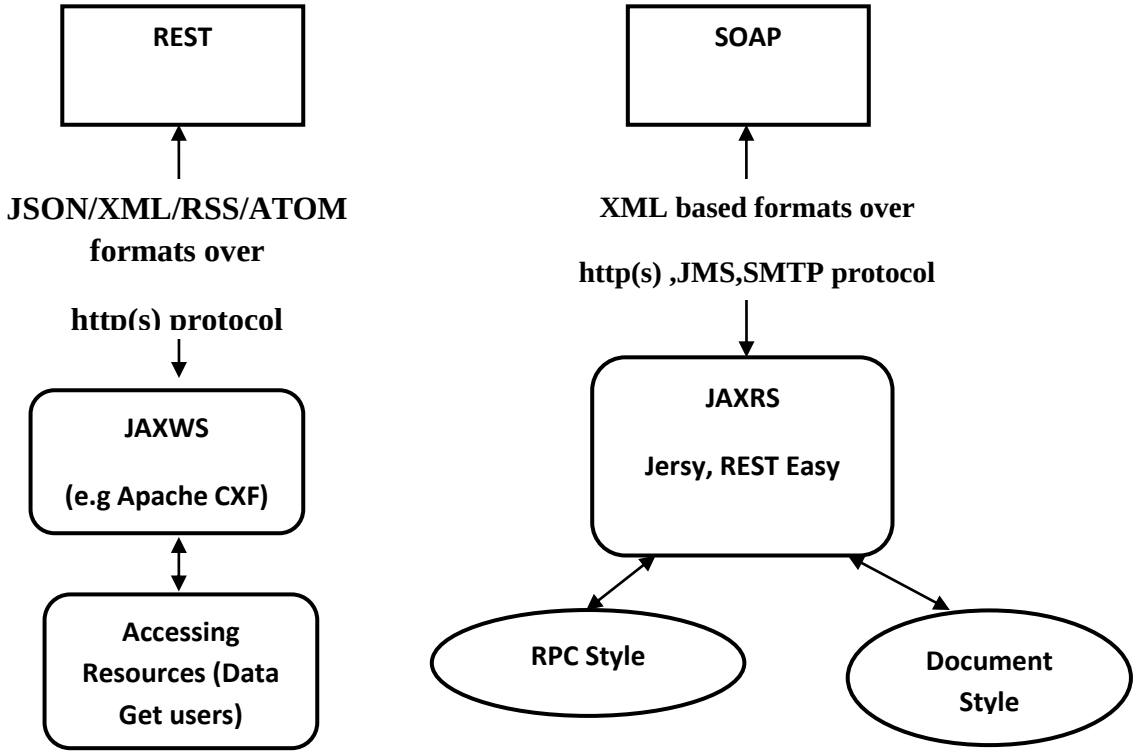


Figure 2.6 SOAP and REST Architectural Difference [228]

2.7 Antipatterns as a quality indicator for SOA technologies

Current and future business requirements change day by day and it is important to meet these requirements and improve the technology for better services.Software architecture is the gateway between business goals and software systems. Quality attributes always related to the designing and developing functional as well as non-functional requirements for end users. Industry is shifting from OOSE to SOA and similarly the quality of SOA system is under consideration.

Antipatterns are used to assess the quality of software and there are multiple approaches and studies reported for OOSE to assess the quality of software by detecting or removing antipatterns from OOSE. Steve Jones published an article on IINFOQ [196] that highlights the SOA antipatterns by providing their description, causes and their effect on the SOA systems. A similar effort is done by different authors which also relate the antipatterns detection for the improvement in various SOA technologies like SOAP and REST.

The antipatterns for SOAP and REST web services are reported in the literature and different techniques are used for the detection of antipatterns, some are publically available and some are not. Moreover, WSDL antipatterns detection also reported that they are the part of SOA technology but still there is no widely adopted technique which can be used as per the industry standards i.e., code -first or contract first.

There is a huge gap of real time correction definitions available for SOA antipatterns as compared to OOSE. Since we are focusing for REST and SOAP, so still there is no tool publicly available for the correction of antipatterns for REST APIs (which are capturing 70% of the market as per the Figure 2.5).

In this dissertation, we are focusing only on SOAP antipatterns detection based on industry standards as well as REST antipatterns correction with the support of antipatterns evolution history from 2015-2017.

2.8 Discussion

Service-Oriented Architecture (SOA) is widely used by software industry but facing real challenges due to violation of design principals. The SOA uses many different protocols used by various technologies that must fulfill the design and composition attributes defined by SOA governing body. In this chapter , we discussed the quality attributes for SOAP and REST services.

There is too much emphasis on the best practices of SOA technologies, but what about the bad practices and their correction? This problem provides a base for this thesis and motivates us to move forward for the detection and correction of SOA antipatterns specifically for SOAP and REST services.

Chapter 3 Systematic Literature Review

3.1 Introduction

Software systems are becoming increasingly complex due to the frequent changes in user requirements. A number of approaches were adopted in the past to address the complexity of software systems and to implement user requirements using structured and object-oriented software development. Object-oriented (OO) software development focuses on the principles of modularity and reusability. Flexibility is obtained via good quality OO design and standardized solutions, such as design patterns [1]. Design patterns are implemented according to the software requirements that are part of software architecture. A software architecture describes software elements and their relationship [19] at different levels of abstraction and with various forms, like classes and methods (OO) or services and servers [123], [124]. Figure 3.1 shows the relationship of software architecture with the OO and SOA.

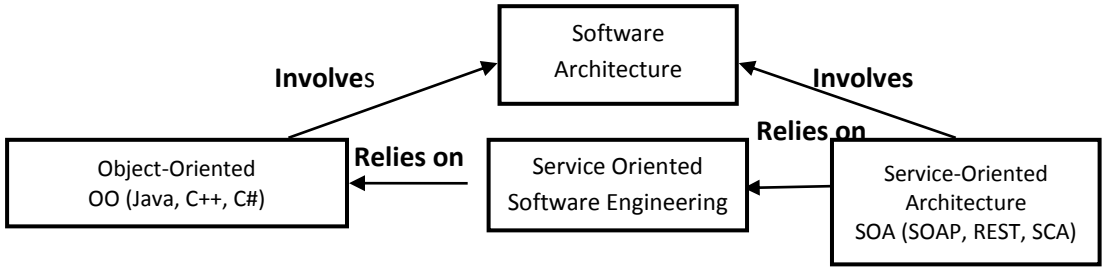


Figure 3.1: Software Architecture with Various Domains

SOA deals with and composes services while the OO paradigm helps the developers to complete the architectural requirements of distributed software [125]. SOA is a software engineering paradigm that provides the foundation for the development of low-cost, rapid, and simple components for distributed environment [18], [125]. SOA relies on different types of services and may depend on OO code to build complex applications [18]. The static behavior of OO code and dynamic nature of SOA is a challenge when maintaining the quality of service (QoS) of SO systems [18].

Software requirements can be functional and non-functional. Although, software developers are usually well-aware of functional and non-functional requirements, they tend to ignore proper design guidelines during the various stages of software development under time pressure. This may lead to the introduction of smells in their systems. Software smells are poor solutions to coding and design problems. Smells are most likely being introduced in software systems because the developers have

little to no idea of the system under design or they have very limited time to implement good design and coding practices. As the system grows, smells may lead to serious problems related to maintenance and technical debts [128]. Fowler and Beck defined 22 code smells in OO and their various refactoring solutions [129].

A recent study explains well the trade-off between delivering acceptable, but immature systems under the constraint of *shorter time to delivery* [130]. This study also investigates why and when the code starts to "smell bad". Releasing immature systems may lead to maintenance problems for the systems [134]. A high-quality documentation is also recommended to achieve and maintain design principles that may prevent and–or remove smells [136]. Other studies report experiments performed to assess the impact of good design on code comprehension [137], the effect of team size on source code quality [138], and human judgments on source code [96], while studies classify smells into code smells, design smells, architectural smells, and antipatterns. Smells are often reported as antipatterns [26, 27] or as code/design smells [24], which opens a debate for researchers to have a consensus on the categorization of smells.

A number of studies reported antipatterns and code smells in different types of services, *e.g.*, antipatterns in SOAP services and WSDL (Web Service Description Language) interfaces [47], [48], [49], in REST [56], [57], and in SCA (Service Component Architecture) [66]. The datasets used by these studies for their experimental validations relied on OO code. In addition, the studies performed by Mateos *et al.* [48], [49] use contract-first and code-first techniques for the detection of smells from the Web service registries.

To the best of our knowledge and from our search in different scientific databases, no previous study classified and compared all the approaches for smells detection across various paradigms, like OO and SOA. As the approaches used for the identification of smells in SO systems are also based on static and dynamic source code analyses and, thus, they may use approaches associated with OO smells identification. A detailed discussion on all the existing approaches is redundant here and out of the scope of this paper because reviews and comparisons of various detection approaches already exist in the literature [28, 29, 30, 61]. We are interested in the evolution of the OO approaches used for the identification of antipatterns for SOA systems. We want to

show the connection between the two paradigms to provide research directions for the reuse of OO approaches for the identification of smells in SO systems.

However, previous studies selected smells based on a single term query, like ‘code smell’ (*e.g.*, [29, 30]) or refactoring opportunities for code smells (*e.g.*, [28, 61]). Therefore, an exhaustive discussion of all types of smell is missing in the literature. We want to provide a complete and exhaustive state of the research on smells. Thus, it is essential to cover all terms associated with smells like ‘code’, ‘design’, 'architecture', and ‘antipatterns’. Eliminating any terms for the selection of primary studies may not fulfill our criteria, as we define in research string, regarding the catalogs of smells reported by researchers in previous studies. Consequently, we carry and report a systematic literature review that focuses on the smells and their evolution in OO and SOA and on the research trends that are well covered in OO approaches but received less attention in SOA. We also report existing research gaps both for OO and SOA smells. We gather and analyzed a set of 78 highly relevant studies addressing six state-of-the-art approaches for the detection of smells.

We extend the scope of this review for OO and SO systems by also including correction approaches after the year 2014, thus complementing a previous systematic literature review (SLR) that reported on the correction of OO smells until the year 2014 [61]. Studies that focus on the refactored pieces of code are out of the scope of our SLR [9, 22] because in this study, we only consider the research works that deal with the identification/detection of smells. This SLR assists researchers and practitioners investigating the issues that received less attention in the literature regarding OO and SOA code smells and may lead to a new research trend by shifting research direction for undiscovered smells that can be detected applying various existing techniques to smells that may require techniques novel for both OO and SOA.This SLR will also help new researchers comprehend the smells and the various techniques reported on their detection.

Table 3.1 summarises the existing systematic literature review for OO and SO paradigms.

Table 3.1: Overview of Existing Reviews

Ref	Time Span Covered	BS Domain (General)	BS Domain Specific	Studies Reported	Review Method	Focus
[30]	1999-2015	CS	OO	46	SLR	Tools, techniques, language used by tools
[33]	Up to 2011	CC	OO	213	SLR	Method, Tools for clone detection
[34]	Up to 2007	CC	OO/SPL/AOS	Exact Studies not reported	Literature Survey	Taxonomy of clone detection techniques and tools
[35]	2000-2010	SWF	OO	36	SLR	Fault prediction in units of software systems
[41]	2010-2012	SPL Smells	SPL	74	SLR	Proposed techniques for SPL
[42]	1996-2003	BS	OO	Exact studies not reported	Literature Survey	Refactoring activities and their roles are discussed
[43]	2000-2009	CS/DS	OO	46	SLR	Methodological, empirical contribution of code smell *w.r.t.* the refactoring
[44]	2001-2012	BS/DS/AP	OO	94	SLR	Model-driven approaches to smells and their effects on model quality
[61]	2001-2013	BS	OO	47	SLR	Refactoring activities and opportunities
This Study	2000-2017	BS/CS/AP/ DS/AS	OO/SO/REST/ SOAP	78	SLR	Focus on smell's evolution, state-of-the-art approaches and research trends in OO and SO

*Code Smells (CS), Design Smells (DS), Architectural Smells (AS), Antipatterns (AP), SPL (Software Product Line), SWF (Software Fault)

As Table 3.1 shows, existing reviews discuss either refactoring approaches or detection approaches. The search terms associated with these approaches mostly

based on code smells. No such review discussed detection techniques and their evolution that help novice researchers investigate smells for SO systems. Moreover, refactoring approaches also focus either techniques used for refactoring [61] or modelling techniques used for refactoring. Furthermore, research regarding the impact of smells on different issues like maintenance, fault-proneness, and lexical impact of code is uncovered. All these areas are comparatively new and mostly reported after the year 2013 for OO and SO systems. Most of these reviews focus code smells, and do not consider the state-of-the-art techniques for architectural smells and antipatterns. This may give the reader an incomplete review that discovered some smells reported as architectural smells or antipatterns. We are unable to find any review that focuses on SO systems and techniques used for those that also evolve in OO and SO paradigms. Previous studies have focused on classifications, but do not discuss research trends that may help new researchers to initiate investigations on these smells.

3.2 Research Methodology for SLR

This SLR reports the existing state-of-the-art approaches on smells from different software engineering paradigms. Kitchenham suggested software engineering researchers apply evidence-based software engineering [45]. The evidence-basedresearch was primarily introduced in the medical domain because expert opinion-based medical service is not as reliable as compared to advice-based health care services. In addition, to collect all relevant facts on research questions, performing an SLR may also help practitioners to find existing research gaps. We follow the guidelines proposed by Kitchenham [45] to perform this SLR in three main steps: planning, conducting, and reporting as shown in Figure 3.2.

This section describes the protocol we follow to perform this review. We also ensure to reduce the chances of search bias. The protocol includes the selection of most appropriate research questions, rules for the study selection criteria, identification of different studies, classification of studies, classification of dimensions for the attributes, and, finally the results of data extraction and analysis.

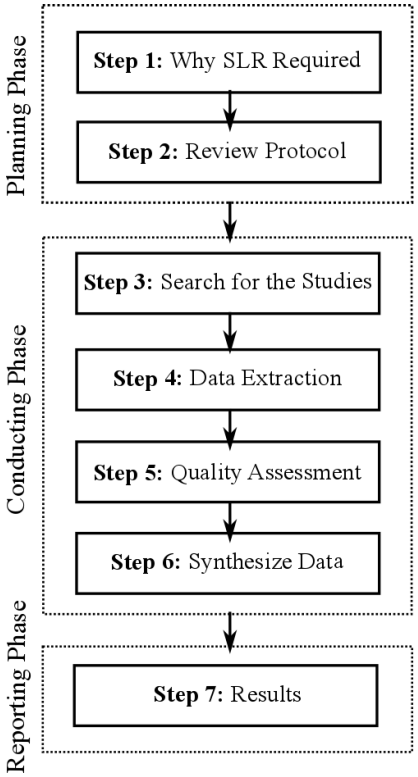


Figure 3.2: Steps Followed for Systematic Literature Review.

Planning the SLR

The main goal of evidence-based software engineering is to collect the most relevant evidence from research and investigate the findings of evidence to evaluate research problems. The state-of-the-art smells techniques are evolved in OO and SO paradigms. However, none of the previously published reviews are similar to the review presented in this study. Most of the reviews were based on code smells from the OO paradigm. In this SLR, we used the following terms to search the primary studies.

((‘*smells*’ OR ‘*code smells*’ OR ‘*design smells*’ OR ‘*architectural smells*’ OR ‘*antipatterns*’ OR ‘*antipattern*’ OR ‘*Antipattern*’ OR ‘*Antipatterns*’) AND (‘*OO*’ OR ‘*services*’ OR ‘*SOC*’ OR ‘*SBS*’ OR ‘SOA’))

The search string is searched from keywords, abstract, and title of each study from the year 2000 until December 2017. Table 3.3 reports the results of our search.

***Review Protocol*:**In the following, we show the general criteria followed in this study to provide more consistent and focused review. We specify the research questions with the help of PICOC criteria [114]:

- *Population:* Object-oriented software engineering, service-oriented computing, service-oriented systems, services, REST, SOAP, WSDL;
- *Interventions:* smells, design smells, architectural smells, code smells, antipatterns, antipattern, antipatterns;
- *Comparison:* A holistic comparison of the population to analyse the impact of recent research on smells, solutions, methods, and techniques;
- *Outcomes:* A classification of state-of-the-art smells techniques that are used to identify or correct smells across paradigms;
- *Context:* An exclusive focus on evidence collected from the state-of-the-art techniques on smells.

Through this SLR, we try to answer five research questions as stated in Table 3.2.

Table 3.2: Research Questions

ID	Research Question	Motivation
RQ1	What are the classifications of the state-of-the-art techniques employed in the detection of smells?	Identification of smells detection techniques followed by their classifications.
RQ2	How the State-of-the-art Approaches Evolved across Different Paradigms Starting from Object Oriented to Service-oriented?	Evolution of specific techniques in OO and service-oriented systems.
RQ3	What are the smells that are reported for a specific paradigm?	Identification of unique smells for a specific paradigm.
RQ4	What is the correlation between smells across the paradigms?	Smells that are repeatedly reported for different paradigms.
RQ5	What are the trends in research for smells from the year 2000 to 2017?	Research trends followed in the domain of smells.

Conducting the SLR

This section presents the review protocol required to perform our SLR. We search for the relevant literature to conduct the SLR.

Search Process for Studies

An effective search string is essential to select the most relevant studies. There is no such clear consensus on the types of smell as design, code, and antipatterns. Therefore, we first go through the relevant reviews presented for smells to avoid any overlapping and then we expand this review for OO and SO systems. We also check the most relevant keyword for the review and check their synonyms, hyponyms, and

alternatives. We rely on the boolean operators like 'AND', 'OR', and the wildcard characters (*) to formulate our search string. As we want to cover all types of smells starting from term code smell to design smell, then to architectural smell, and, finally, antipatterns, therefore, we use each search term associated with smells starting with the help of the wildcard character (*) and 'AND' operators to include the relationship between population and intervention. General terms related to smells were searched from different digital libraries along with keywords and full term-based search. Table 3.3 reports the result of each term associated with smells.

Table 3.3: Number of Studies Found in Selected Digital Libraries after General Term Search.

Sl. No	Terms Search	IEEE	ACM	Science Direct	Wiley	Springer	Total
1	Code smell, code smells, code flaws	195	1,586	86	1	32	1,900
2	Antipattern / antipattern / antipatterns	48	1,719	125	0	1	1,893
3	Design smell, design smells, design flaws	135	6,550	84	1	8	6,778
4	Architectural smell / architectural smells	27	2,048	382	0	0	2,457
5	Smells	58	54	100	49	480	741
Total		**463**	**11,957**	**777**	**51**	**521**	**13,769**

Study Selection

To select the most relevant research studies, we applied a three-step process:

Step 1: We extracted 13,769 studies resulting from the generic keyword-based search strings from different digital libraries. The keyword-based search also reports the articles from requirement engineering and performance antipatterns. Initially extracted studies are further refined for the domain of software engineering resulting 2,669 studies left in the pool for review. The majority of the studies are removed from the ACM library because terms associated with the research strings are also available in the domain other than software engineering.

Step 2: The collection of studies selected in Step 1 is further refined manually by covering index terms, abstract, title, and their application domains (OO and SO). This process removes all studies from the domain of requirement engineering and Android applications containing various terms related to smells. Duplication is removed

among research studies from the selected databases. The resultant provided 540 studies out of 2,669 based on their matching definitions of smells related to design, code, and architecture.

Step 3: Furthermore, studies are filtered following some *exclusion* and *inclusion* criteria. Only the studies from well-known conferences are kept and the rest is discarded. The *inclusion* criteria are based on the following:

- Journal articles are selected related to the domain of object-oriented analysis, software maintenance, reverse engineering, information and software technology, service-oriented architecture, and Web service;
- Top-level conferences are selected when related to software maintenance, reverse engineering, object-oriented technology, evidence-based software engineering, and service-oriented computing;
- In this SLR, we include all studies associated with the term smells (*e.g.,* code, design, antipatterns, and architectural smells) ;
- Contextual data for each study is provided in the Excel sheet available online[31].

Whereas the *exclusion* criteria include:

- Articles of short length (less than five pages);
- Book chapters are not included;
- Workshop articles and lecture notes are not included;
- Software performance antipatterns and software requirement antipatterns are not added due to their irrelevancy to our target domain as we are working the evolution of smells in OO and services and requirement and performance could not be part of the evolution from one system to another;
- Research works published as a technical report;
- Research studies related to code clone, duplicate code, and copy-paste programming is not added since reviews exist for them;
- Smells related to android systems are not added since in this SLR we are covering only the paradigm of OO and SO;
- Research studies that discussed single smell are also not added because we want to know which smells are mostly discussed by tools, industry, and academia.

After applying Step 3, only 75 studies are left that satisfy the above-mentioned inclusion and exclusion criteria. Finally, snow-balling method [110] is applied to check the reference list of the selected studies to minimize the chance of removal of any relevant studies. Therefore, in an additional activity, 78 studies are selected in Step 4. The snowballing provides additional three studies that are [83], [98], and [23], mostly cited in different research studies and not included in selected searched databases. Figure 3.3 shows the representation of studies selection criteria.

The identification of smells is performed following an incremental process. In the first phase, we start with a primary study and collect information on all reported smells. We then follow the process across all the primary studies for different domains, and, finally, we get a pool of smells for a specific domain. In the second iteration, we run the process for identification of smells and check whether these smells are already 'reported' or 'detected' or 'corrected' in the area other than OO, if yes, then we add those smells into correlated smell section to check what type of smells are evolved across paradigms. If we are not able to find that smell as 'corrected' for SO systems, then we report this smell as 'not corrected'. Similar iteration is followed for techniques evolved in OO and SO to check the approach followed in a sequence from the year 2000 until 2017. This iteration will also help to identify trends in the research on smells.

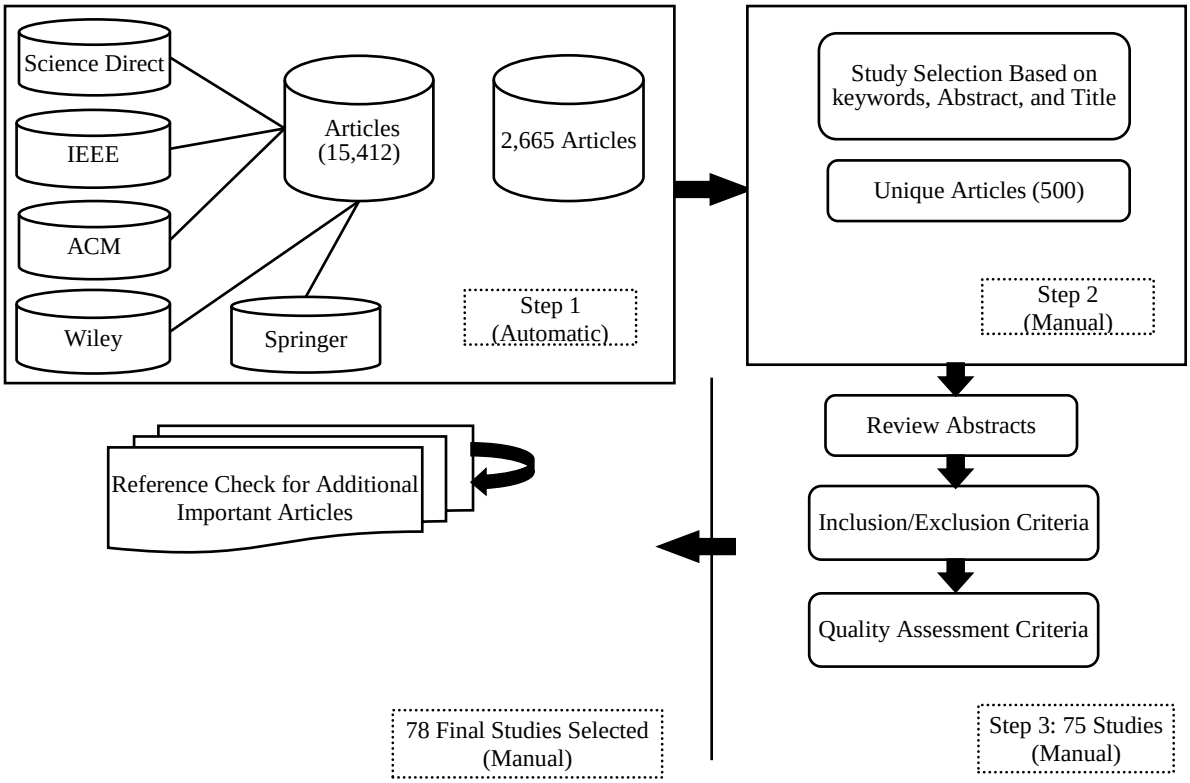


Figure 3.3: Study Selection Criteria

Data Extraction and Analysis

We extracted data in Excel in a consistent format as presented on our online appendix[31] where we also present detailed results. Data is extracted based on the research questions. We focus on the types of smells, *i.e.*, code, design, architectural, and antipatterns, as they appear in the title, abstract, or index terms of an article. Our classification technique is based on static, dynamic, empirical, methodological, and linguistic source code analysis. Research trends are collected after examining the sequence of related research patterns over the past seventeen years. Data extracted is evaluated and disagreement was discussed until the conclusive results are achieved. Many selected studies did not answer all the questions available in our data extraction form. Table 3.4 presents the data extraction sheet designed for each research question.

Table 3.4: Data Extraction Sheet.

Search Criteria	Data Item	Description
General	Identification number	Reference Number Assigned to the article
	bibliography	Year, Title, Source/Research Group
	Type of the article	Conference Paper, Journal Paper/Tech Report
	Study aims	Summarising notes about each study
	Study designs	Experimental, Case Study, Survey, Review
RQ1	Behavioral source code analysis	Source code analysis that uses Source code metrics to examine soured code behavior
	Dynamic source code analysis	Analyse the interrelationship of program entities after the execution and checking the behaviour of the program
	Algorithm-based analysis	Studies that use a specific algorithm to detect smells from source code
	Empirical source code analysis	Studies that report the results of already established tools to empirically evaluate the research problems and address some new findings
	Methodological analysis	Implements already proposed a methodology in a new way to either correct or detect smells. These types of analysis compare the results before and after the implementation of any specific method.
	Linguistic source code analysis	Checking the internal code quality like naming conventions of methods, classes, etc.
RQ2	Evolution of research for smells in OO and SO	Analysis of multiple research methodologies constantly repeated from paradigm to paradigm.
RQ3	Smells reported for a specific paradigm	Unique identification of smells for OO, SOAP services, REST, SCA, AOP, SPL

RQ4	Evolution of smells in OO and SO	Identification of smells that are reported for OO, but also later found in SO systems
RQ5	Trends in research	Unique trends for Smells Detection Correction Maintenance etc.

Data Synthesis

Quality criteria are based on the inclusion/exclusion criteria as defined above. Metadata analysis is performed after reviewing the studies completely. Metadata analysis for each research question is clearly examined and the answer is recorded in an Excel sheet as presented online[31]. We have verified basic contextual information reported for each research question. A number of approaches are available for data synthesis some of which maintain the qualitative form of the evidence such as meta-ethnography while some involve converting qualitative findings into a quantitative form such as content analysis [111]. Basic quality criteria for selecting studies discussed above are based on the guideline provided by Kitchenham [46]. There are different terms reported for the smells in the literature but most of the smells are reported as code smells (22 research studies), and only nine studies are reported as antipatterns. Data addressing our five research questions are extracted from the 78 most relevant studies that satisfy all the quality criteria, including the PICOC and the inclusion/exclusion checklist. Our goal is to collect most relevant data from the studies selected to analyse the state-of-the-art approaches in OO and SO paradigm. To investigate the key questions, three sets of data were extracted from 78 studies.

i) *Context data* showing the context of each study, such as the source of data, experimental evaluation, application area, and programming languages are noted;

ii) *Qualitative data* related to research questions were extracted from the index terms to assess the type of smells *w.r.t.* the term generally associated;

a) *Qualitative studies* through the extraction of qualitative data provides information for the studies that used cause-effect relationship or introduce new ideas by using different properties of the system under analysis, *e.g.*, studies [77] and [88] predict certain properties of the system or studies [30], [61] use

different attributes in terms of research questions as qualitative measure to show their findings.

iii) *Quantitative data* extracted from the studies based on the predictive performance of the model or approach reported in the study. The data are divided categorically and the variable used to represent result is mostly continuous. However, some of the studies reported their results in both forms:

a) *Categorical studies* report their results predicting whether the smells detected, corrected, or maintained in the system under analysis. These results are reported by using accuracy measures like precision or recall. In total, 28 studies used accuracy measures like precision and recall in this SLR.

b) *Continuous studies* reported their results by using similar measures like mean standard error (MSE) or measures the difference between expected and observed results like chi-square, correlation, logistic regression, and ranking form. We found 21 studies that come in this category as they report their results by using statistical techniques to validate their research model and present their findings. The most widely used technique for continuous studies is correlation analysis (six studies) followed by regression analysis. The complete list of studies falls into this category is given below in Table 3.5 along with the techniques applied.

Table 3.5: Techniques for Continuous Studies.

Ref. No	Technique Used
[38]	Fisher Test
[51]	Mean, Median, SD, Correlation
[49], [53], [91]	Correlation
[68], [73]	Wilcoxon Rank Test
[78]	Correlation
[81], [113]	Chi-Square Test
[77]	Regression Analysis
[80]	Cliff's D and Kruskal Wallis test, Holm's Mann
[98]	Cohan's Kappa, Fleiss Kappa
[104]	Correlation Analysis, Regression

[105]	Wilcoxon Rank Test, Mean, Standard Deviation, Median
[104]	Logistic Regression Model
[15]	Min, Median, Mean, Mode
[59]	Proportion, Odd Ratio
[84]	PCA, Logistic Regression
[23]	Logistic Regression Model, Odd Ratio
[119]	Fisher test, Odd Ratio, Chi-squared

The distribution of studies *w.r.t.* the types of data they used are given in Figure 3.4.

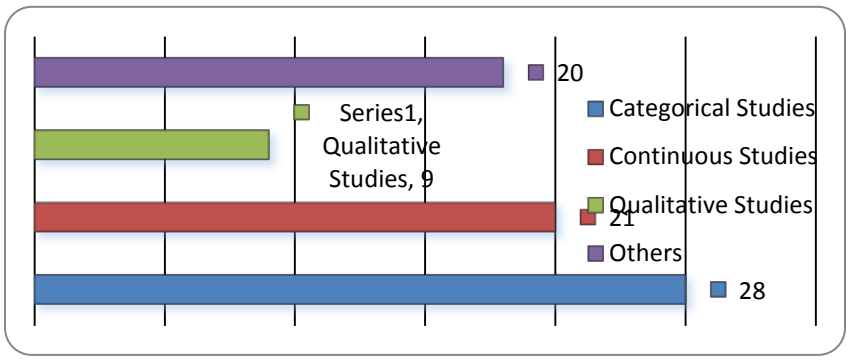


Figure 3.4: Distribution of Studies w.r.t. the Data Analysis Techniques

It is important to consider that 22 studies are reported as 'Others' because they used mixed approach and did not report their results by using any measures. These types of studies have mostly focused on the key concepts that are proposed but were not tested or validated. The complete information about these studies is provided online[31].

Data synthesis is combined with the data extraction form to analyse the quantitative as well as qualitative data fully. Data extraction form as reported in Table3.4 provides complete information regarding each research question along with the data synthesis reported in Figure 3.4. The information presented helps to look for the most applied statistical methodology used by the industry for categorical, continuous, and qualitative studies. In total, 18 studies used a mixed approach based on the quantitative and qualitative information. Some of them only discuss the concept or novel approach and present findings or benefits of their approach.

Table 3.6: Frequency of Smells Terms in Studies

Sl.	Smells Type	Frequency	%	Reference

1	Antipatterns	11	14.1%	[38], [58], [63], [23], [76], [88], [101], [106], [119], [120], [121]
2	Architecture	15	19.23%	[47], [48], [49], [50], [51], [52], [53], [54], [56], [57], [66], [67], [68], [100], [102]
3	Unpleasant smell	7	8.75%	[78], [82], [90], [93], [94], [96], [98]
4	Code smell	24	30%	[60], [61], [30], [69], [70], [73], [112], [17], [74], [84], [77], [79], [85], [86], [91], [12], [97], [99], [103], [104], [105], [114], [115]
5	Design smell	18	23.5%	[26], [62], [65], [16], [15], [25], [71], [72], [75], [27], [81], [83], [87], [89], [92], [80], [64], [59]
6	Code and Design	3	3.85%	[54], [2], [95]
Total		**78**	**100%**	

Moreover, it is also observed that most of the studies have validated their research model on open-source systems. Therefore, most of the results in the area of smells can be compared or tested by analyzing similar open-source systems. Figure3.5 reports 66 primary studies that use open-source systems and three other studies that use propitiatory systems, *i.e.*, other than open-source systems. We also found nine studies that do not rely on any target systems, open-source or proprietary, to validate their results.

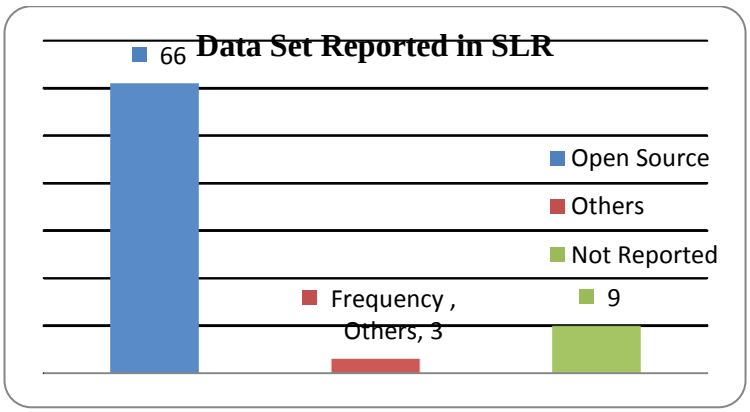


Figure 3.5: Data Source Used for the State-of-the-Art Research on Smells

The validation criteria reported by the studies are either based on accuracy as a measure or by using statistical techniques. We have found the highest number of research studies between the year 2013 and 2015 (18 studies). Moreover, the research trend has been moved from static source code analysis to dynamic source code analysis applying machine learning, artificial intelligence, and genetic algorithms. Furthermore, smells detection has not gained much attention for SO systems. Figure 3.6 shows the distribution of studies over the years. Fowler [8] introduces the concept

of code smells in 1999 and the first paper reporting smells published in 2001 [14] followed by four studies in the year 2004 and two studies in 2006. These are conference studies and we are unable to find any journal studies in those years that fulfill the criteria of selection for studies. The problem of identifying smells in source code began to attract more research attention in the year 2010 with an average of more than six research studies per year. This observation highlights the interest of researchers and the importance of smells after the year 2009.

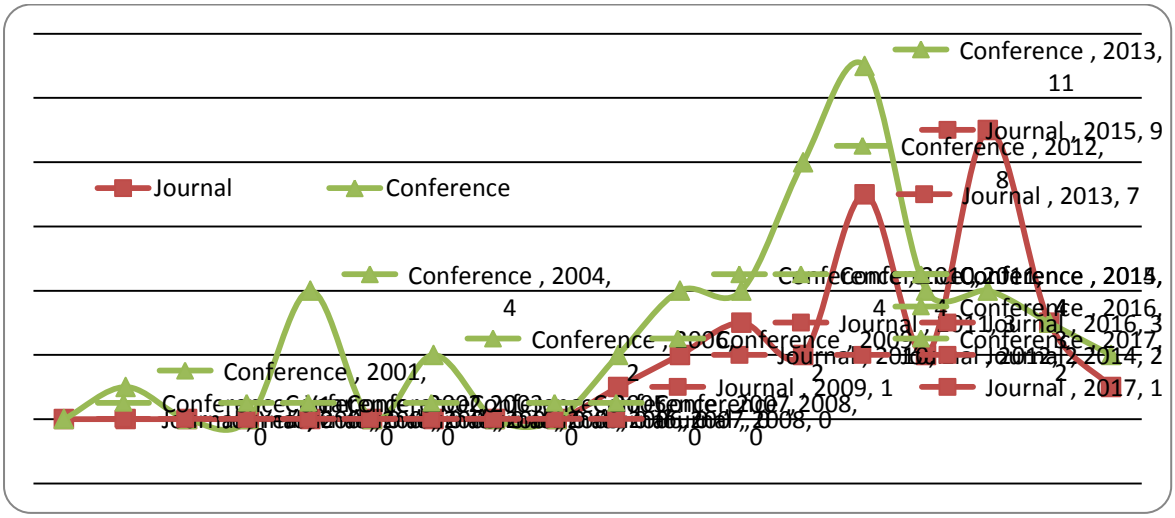


Figure 3.6: Year Wise Distribution of Studies.

The research studies were published in 39 different venues. Over half of the studies are published in conferences and the rest are published in various journals. A slight shift from conference to journal articles shows the importance of considering both conference and journal articles in this systematic review. Also, considering either journal articles or conference articles will create a research bias and may provide the readers with an incomplete literature review. Furthermore, researchers are attracted more towards the conference of reverse engineering, software maintenance, whereas, for services, the SO computing attracted more researchers.

There is less number of studies reported for service-oriented software engineering compared to OO. This shows that research is now shifting towards SO systems due to the high demand for Web services. Most of the studies for SOAP services are reported in the International Journal of Web and Grid Services (four studies) and IEEE Transaction on Service Oriented Computing (one article). Moreover, we are unable to find any journal paper on REST services smells as well as correction of smells for REST services. This observation shows that this is a highly active area of research for

the new researchers. Figure 3.7 presents studies over the past 16 years for the paradigms of OO and SO.

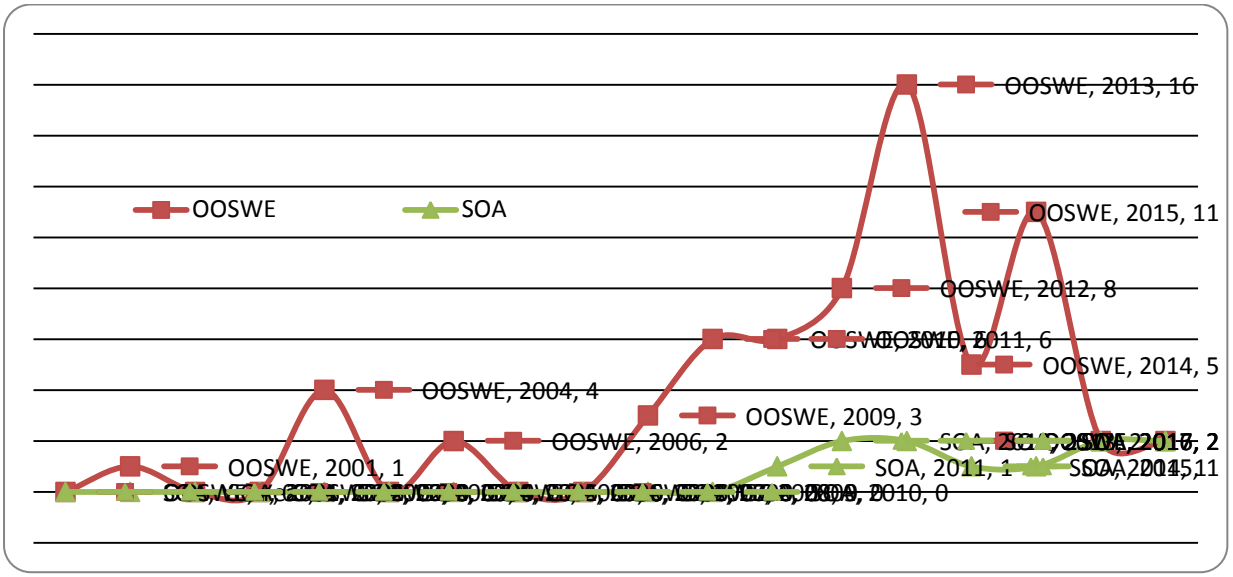


Figure 3.7: Year Wise Distribution of Studies w.r.t. Different Paradigms.

The authors of all the above studies are from academia and mainly working in research groups with support from industry. Therefore, based on this study's selection criteria, no strong evidence was found that gives strong implication whether research on smells is primarily conducted by the industry or the academic community. We also found some references where academia solves the industry problems after collecting information from industry blogs like J2EE[1] and INFOQ[2] [47, 57, 52] and provide tool support that solves the reported problems by industry blogs.

3.3 Classifications of the State-of-the-art Techniques Employed in the Detection of Smells

The classification of the detection techniques for smells is also reported in a recent review [30] but it is focused only on the smells discussed by Fowler [8]. This focus creates a bias because smells other than Fowler's are not reported. Moreover, the selection of studies covers from January 2000 until December 2017.

Another systematic literature review used the term "smell" for selecting studies, but focused only on studies that discussed multiple refactoring operations [61]. It focused only on the refactoring for removing smells and not on the impacts of smells on system performance and multiple approaches used for the detection of smells. We

1 j2ee.com

2 infoq.com

also found studies that discussed model-driven approaches for smells until the year 2011 [41] due to their research associated with the detection of smells. Moreover, we also discussed studies that introduced correction approaches in addition to the detection techniques that were not considered in the most recent review [30]. We divided the studies on the basis of source code analysis and not based on the symptoms associated with smells like reported in [30]. The classification is divided into (1) static source code analysis and (2) dynamic source code analysis.

Static source code analysis is a technique that examines the properties of smells and their impact without executing systems. In contrast, dynamic source code analysis examines the cause and effect relationship of smells during system execution.

In the following, we classify primary studies selected for this review into two major types: (1) dynamic source code analysis and (2) static source code analysis. Most of the studies focused on static source code analysis as we found 70 primary studies under this category compared to 8 for dynamic source code analysis. Figure 3.8 reports the total number of studies reported for each type of analysis.

3.3.1 Static Source Code Analysis

We further divide the static source code analysis classification into six categories as shown in Figure 3.8.

a) Behavioral Source Code Analysis

Software systems undergo various changes and study [9] reports that 98% of the literature on change impact analysis is related to code analysis in comparison to 17% of the studies to architectural changes. We classify multiple primary studies into *behavioral source code analysis* that examine the program behavior without executing the source code. The behavioral source code analysis uses different metrics to check program behavior like cohesion, coupling, depth of inheritance, lines of code using various source code metrics.

The detection of smells is not possible without any intermediary representation. This representation is used to extract useful information from the application and to apply source code metrics to check the detection of smells. Therefore, behavioral source code analysis is based on either informal description of flaws [16] or using the textual description of rules with the help of domain-specific language [24]. Source code parsing is also appliedusing different parsers [49, 51, 53, 100] that make an

intermediary representation of source code and apply source code metrics directly to this intermediary representation. Some code smells are also reported in the literature after examining the version control histories and then apply source code metrics [17]. These types of studies help industry and academia to investigate the problems for a system under analysis to improve its quality.

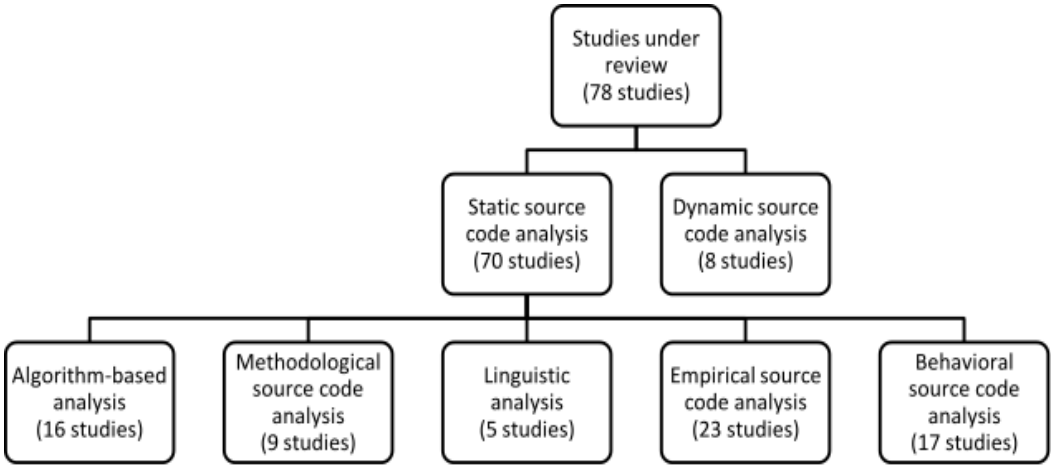


Figure 3.8: Distribution of Primary Studies based on Source code analysis

Table 3.7: Studies Used Behavioral Source Code Analysis

Ref. No	Smell Type	Precondition	Post Condition
[16]	DS	Informal description of flaws	Source code metrics
[17]	CS	Change history extractor using SVN[3], CVS[4]	Code smell detector applies for each smell type
[24]	CS/DS	Textual description is used for DSL to generate detection algorithms	Source code metrics
[55]	CS/DS	Parsing using JFLEX[5] and Java Cup[6]	Source code metrics
[60]	CS/DS	None	Source code metrics
[65]	DS	Meta-model	Source code metrics
[95]	CS, AP	Ontology	Source code metrics
[78], [75], [25], [62], [83]	DS, BS	Smells properties are applied	Source code metrics
[101]	AP	UML specification is defined for each AP	Correction approaches are defined
[49], [51], [53], [100],	AS, AP	Java to WSDL file	Source code metrics

*Code Smells (CS), Design Smells (DS), Architectural Smells (AS), Antipatterns (AP)

3 https://subversion.apache.org/

4 http://www.nongnu.org/cvs/

5 http://jflex.de/

6 http://czt.sourceforge.net/dev/java-cup/

We also observed that meta-model [65] and ontology [95] are other forms of intermediary representations on which source code metrics are applied to detect smells. Moreover, smells identification could be done either specifying them in the form of textual description [78] that helps to improve WSDL document describing SO systemsand also OO systems [25, 62, 75] or in the form of UML specifications [101] to improve the source code quality. Table 3.7 highlights the conditions used by the primary studies for the smells detection. It provides the primary studies reported for OO and SO systems.

b. Empirical Source Code Analysis

Empirical source code analysis has been the technique to get information by using already established tools or approaches either directly or indirectly. Empirical source code analysis is either analysed quantitatively or qualitatively [122]. The evidence collected from different techniques will help researchers to answer questions that are clearly defined and collected from different problem domains [122].

Some smells detection techniques use empirical evidence collected by different already established tools to check the associations among various versions of OO systems [38, 22, 68] or systematic literature review to check the most relevant information presented for either detection of code smells [61, 84, 30] or correction of code smells [64] or code decay [85].

Empirical source code analysis mostly investigates the cause-effect relationship like smell *vs.* maintenance effort [77, 112], best practices of services in open-source platform [120], class performance [104], change-proneness of antipatterns and clones [119], refactoring suggestions [80, 97, 104] after smells detection or statistical techniques to check the relationship between already presented smell and design patterns [105]. Table 3.8summarises the extracted information of 23 studies that use empirical source code analysis techniques.

Table 3.8: Empirical Source Code Analysis Techniques.

Ref.	Smell	Pre-Condition	Post Condition

	Types		
[22]	AP	Mining the source code	12 versions of Eclipse[7] and nine versions of ArgoUML[8] are used to mine the repositories for antipatterns and code smells
[27]	AP	Change-proneness and Null pointer exception in classes are analysed	Results are validated in 11 releases of Eclipse
[38]	AP	DÉCOR[9] along with PTDIEJ [7] tool suite to check static dependencies	Macocha[10]mines version-control systems for checking association among antipatterns
[68]	AS	SO system FraSCAti[11] is used for evaluation purposes	Python script is used to check the commit with changes and without changes
[74]	CS	Mimec [88] is used to record log	12 code smells detected in the pre-maintenance version using Borland together and INCode
[77]	CS	Borland together and INCode[12] for the detection of code smell	Mimec [88] is used to check developer's activity, and then maintenance effort is analysed
[112]	CS	Three developers are hired to perform maintenance tasks	Regression analysis is carried out to measure effort
[23]	CS	Changes are counted using CVS from Eclipse	Logistic regression is applied to correlate the presence of antipatterns with the change process
[80]	AP	Antipatterns are detected, MADMU [109] matrix is used to check the cost	Refactoring options are suggested to remove smells
[81]	DS	Design smells are analysed*w.r.t.* Flaws classes	Design defects *vs.* flaws classes
[88]	AP	DÉCOR is used for the detection of code smells	Odd ratio and Fisher test are applied to check the difference between mutated and non-mutated antipatterns
[106]	AP	A set of guidelines is defined for WSDL	Easy SOC plug-in developed for detection
[91]	CS	Manual analysis of smell frequency and source code metrics for the detection of	Correlation is used to check thefurther relationship

7 https://eclipse.org

8 http://argouml.tigris.org/

9 http://ptidej.net/tools

10 http://www.ptidej.net/

11http:// frascati.ow2.org/

12https://marketplace.eclipse.org/content/incode-helium

		smells	
[69]	CS	Eclipse plug-in is developed to check the location of source code	Naïve Bayes and association rule mining is applied to check the bug relationship in code
[97]	DS	INCODE[13] as Eclipse plug-in to check the quality of the code	Refactoring suggestions for code quality
[98]	CS	Different smells detector is used to check smells in code	Kappa[14] statistics are applied to check results
[104]	BS	Mining the evolution of three open-source project	Quantitative analysis of the classes participating in refactoring
[105]	CS	Nine design patterns and seven code smells are analysed	Correlation, mean, median is used to answer the research problem
[61], [85], [30]	CS	Kitchenham guideline to collect relevant literature	Reported code smells detection and correction approaches
[119]	AP	DECORE for antipatterns detection and CC finder for clone detection	Co-change analysis and fault -proneness identification with Macocha model
[120]	AP	A literature review of services collected	Analysis of Google Cloud platform, Stack overflow, OCCI for best practices

*Code Smell (CS), Design Smell (DS), Architectural Smell (AS), AP (Antipatterns), SC Metrics (Source Code Metrics)

c. Algorithm-based Source Code Analysis

Some smells detection methodsuse more than one detection technique, like source code metrics. Some use genetic algorithms [12, 59, 79], machine learning techniques [26, 93], or image processing [92]. These algorithms help to solve the problem of using fixed threshold for the detection of smells [59] and correction of smell using development history [12]. Moreover, detection results from different repositories are used to implement machine learning techniques [26, 93]. These studies also open an opportunity for the detection of smells and correction using machine learning algorithms with good precision and recall as reported in our online appendix[31]. Table 3.9summarises the information of all studies that use different algorithms.

13 https://marketplace.eclipse.org/content/incode-helium

14 https://en.wikipedia.org/wiki/Cohen's_kappa

Table 3.9: Algorithm-based Source Code Analysis.

Ref. No	Smell Type	Algorithm Domain	Pre-Condition	Post Condition
[26]	AP	ML	60 source code metrics results are used as training datasets from PROMISE[15]	SVM classifier is applied using WEKA[16]
[52]	AS	GA	Base examples are collected from Different Web service search engines	Different combinations of source code metrics are applied
[59]	CS	GA	GA is used for dynamic threshold adaptation	Source code metrics are retrieved from different tools
[58]	AP	-	Perl Script along with Ptidej tool used to compute metrics from 12 versions of Eclipse and nine versions of ArgoUML	DÉCOR is used to detect antipatterns
[70]	CS	-	The parallel evolutionary algorithm is used	Source code metrics thresholds are set using the GA algorithm
[71]	AP	Mathematics	Antipatterns are formally defined	Prolog rules are applied for detection
[76]	AP	CG	A set of metrics and their values are used to generate B-Spline	The similarity of thesignature is computed to detect Antipatterns
[79]	CS	GA	GA is implemented with the help of source code metrics	NSGA-II for the correction of defects
[89]	DS	BMS	Initialization, training, memory selection cell	Source code metrics selection for detection
[90]	BS		Set of reference code for refactoring	GA is applied
[92]	DS	IP	Input is the name of the method, association type among classes	Similarity scoring and Bit vector algorithm applied to check Smells
[54]	AP	GA	The PE-A algorithm is applied	Different combination of best threshold source code metrics for detection
[93]	BS	ML	Source code metrics results from seven different software repositories	Naïve Bayes, Logistic, IB1, IB k, VFI, J48, and random forest applied

15 http://promise.site.uottawa.ca/SERepository/

16 http://www.cs.waikato.ac.nz/ml/weka/

[12]	CS	GA	Source code metrics	NSGA-II applies to check detect and correct smells from development history
[108]	CS	ML	78 systems are studied, and different source code metrics are analysed	Combination of different machine learning algorithms are applied to check the smells for each system under analysis
[121]	AP	ML/GA	Source code metrics are used	An evolutionary algorithm for antipatterns detection

*Code Smells (CS), Design Smells (DS), Architectural Smells (AS), AP (Antipatterns), ML (Machine Learning), IP (Image Processing), GA (Genetic Algorithm), BMS (Bio Medical Sciences).

d. Methodological-based Source Code Analysis

Methodological-based source code analysis is used to implement an existing methodology in an alternative manner for either detecting or correcting smells with the end goal of improving detection accuracy.

This type of source code analysis can be used to define unique technique relying on already available tools or algorithms to detect occurrences of smells. The studies in this aspect mainly focus on the quality of the source code before and after implementation of specific methodologies, like performance comparison of different queries using some source code metrics [63] or using different tools for smells detection [99].

Moreover, NSGA-II (Non-Dominated Sorting Genetic Algorithm) algorithm is also used to check source code quality before and after applying algorithms for the correction of smells [102]. The effect of smells on system defects was also studied by considering five types of smells [114]. This study showed that the *switch statement* has more influence on defects in comparison to other smells under study. The studies reporting empirical source code analysis are presented in Table 3.10.

Table 3.10: Methodological Source Code Analysis.

Ref. No	Smell Type	Precondition	Post Condition
[63]	AP	Execution and transformation of queries to make ASG	Create an EMF representation of the ASG for performance comparison
[99]	AP	J-Deodorant[17], Check style[18] and InCode is used for the detection of smells	Extract Class, Encapsulate Fields and Move Method refactoring is applied using Jason 1.3.10[19] and Eclipse Kepler[20]
[102]	AS	Source code metrics and Multi-objective optimisation approach	NSGA-II implemented to check source code quality before and after refactoring
[50]	AS	Migration strategies are defined for Legacy application to SO systems	Refactoring of WSDL document is applied combining different thresholds of SC metrics
[72]	DS	CLIO approach used to detect structural and change coupling	Modularity violation calculated by comparing structural and change coupling
[86]	CS	The tool is developed that run in the background to monitor changes	Monitor invokes smell detection tool and warns developers
[82]	BS	Pattern-based definitions are presented based on the symptoms of smells	A survey is conducted to get consensus on revised and improved definitions
[96]	BS	Two parts Web-based questionnaire is developed to get anopinion from developers about smells	Different options are analysed like an evaluation of developer perception, demographic effects, and experience of developers *w.r.t.* the code smells
[114]	CS	Negative binomial regressions is run to check the faults in investigated systems	Different suggestions are passed that helps researchers for refactoring

*Code Smell (CS), Design Smell (DS), Architectural Smell (AS), AP (Antipatterns), EMF (Eclipse Modelling Framework), ASG (Abstract Syntax Graph), SC (Source Code).

e.Linguistic Source Code Analysis

This technique of source code analysis through linguistic quality assessment started early 2015. There are only three studies published in the OO paradigm that investigated linguistic antipatterns. These antipatterns are erroneously introduced in

17 https://marketplace.eclipse.org/content/jdeodorant

18 checkstyle.sourceforge.net/

19 https://sourceforge.net/projects/jason/files/jason/version%201.3.10/

20 www.eclipse.org/downloads/packages/release/Kepler/SR

the code when using wrong naming conventions of methods, classes, and variable names. The detection of linguistic antipatterns is a new area of research receiving growing attention in the software engineering research community. We were unable to find any study that reports the effect of linguistic antipatterns on system performance. Moreover, it is required to implement correction approaches for linguistic antipatterns.

Table 3.11: Linguistic Source Code Analysis.

Ref.	Paradigm	Pre-condition	Post-condition
[15]	OO	1) Check methods, attributes, leading comments using Stanford Natural Language parser 2) The semantic relation is analysed using WordNet[21] and implemented as Eclipse plug-in	Seven open-source systems archives are used to check linguistic antipatterns using LAPD, then online questionnaires are designed to check the developers' perception towards linguistic antipatterns (LAP)
[64]	OO	Linguistic antipatternsare defined	Case examples are given to analyse the LAP
[57]	SO	Syntactic and semantic similarities are studies using Stanford Parser[22] and WordNet lexical database	DOLAR[23] tool is developed to detect linguistic antipatterns from REST API
[59]	OO	Lexical and design smells are detected in 30 releases of three projects: ANT[24], ArgoUML[25], Apache[26]	Fault-proneness is checked for design smells *vs.* lexical smells
[84]	OO	Structural metrics are applied	Principal Component Analysis along with different statistic measures to evaluate the subject system

These techniques used several types of parsers like Stanford Natural Language parser to detect parts of speeches. The LAPD [15] tool is proposed for the detection of linguistic antipatterns based on the NLP parser to detect similarity between class names, variables, and methods by manually implementing smelldetection techniques to detect linguistic antipatterns. A recent study reported the effect of linguistic antipatterns on change-proneness [64]. Table 3.11 highlights different conditions used for linguistic source code analysis.

21 https://wordnet.princeton.edu/

22 http://nlp.stanford.edu:8080/parser/

23 http://sofa.uqam.ca

24 http:// ant.apache.org/

25https://sourceforge.net/projects/argouml/

26 http://www.apache.org

3.3.2 Dynamic Source Code Analysis

The smells detection techniques applying dynamic source code analysis techniques mainly analyse the execution states of the systems after their execution under real execution scenarios. Most architectural smells detection techniques use dynamic source code analysis and belong to the paradigm of service-oriented architecture [47, 56, 57, 66, 67] relying on DSL that helps to generate algorithms along with service interface, using FraSCAti[11] runtime support for static, dynamic, and lexical analysis.

The dynamic source code analysis also used a dynamic threshold adaptation instead of fixed thresholds for smells detection [115]. Genetic algorithm along with tuning machine is applied to check the results with inferred settings, the default settings, and with a tuning algorithm [115]. We found eight studies that reported dynamic source code analysis for OO. The complete information about different techniques implemented using dynamic source code analysis is presented in Table 3.12.

Table 3.12: Dynamic Source Code Analysis Techniques

Ref. No	Smell Type	Precondition	Post Condition
[47]	AS	DSL is used along with algorithm generation to map rules	Static, dynamic and lexical source code metrics
[56]	AS	DSL is used along with service interface to invoke the Services by using FraSCAti and Apache CXF[27] runtime support	Wrapping REST API with FraSCAti SCA analysis
[57]	AS	DSL is used along with service interface to invoke the services by using FraSCAti and Apache CXF runtime support	WordNet, Core NLP is used to analyse lexical properties
[66]	AS	DSL along with FraSCAti runtime support	Source code metrics
[67]	AS	Association rule mining to check association among execution of services	Source code metrics
[73]	CS	Detect smells on theclient side	Check detected smells on theserver side
[103]	CS	Mining the source code through SVN	SrcML[28] toolkit and MARKOS[29] code analyser used
[115]	CS	Tuning machine is applied on inferring set to check most appropriated thresholds	Smells are checked and refactored after applying

27 https://github.com/apache/cxf

28 http://www.srcml.org/about-srcml.html

29 http://markosproject.sourceforge.net/downloads/

*Code Smells (CS), Design Smells (DS), Architectural Smells (AS), AP (Antipatterns), DSL (Domain Specific Language), SVN (Sub Versioning Number), NLP (Natural Language Processing), SCA (Service Component Architecture)

Summary on RQ1: Research on smells analyses the target systems by applying source code-level metrics that helpinvestigate systems by using lexical properties. Research in the domain of smells also empirically validates findings using statistical measures after investigating the cause-effect relationship with some independent variables, like the numbers of defects. A recent shift towards the use of different algorithms from machine learning as well as artificial intelligence also helps to detect design smells and may improve the performance of detection techniques. These algorithms are reported for the detection of smells both in OO and SOparadigms. We have identified a few studies that reported the use of lexical analysis (*e.g.*, [15], [59]) and dynamic source code analysis (*e.g.*, [57]).

3.4 Evolution of State-of-the-art Approaches

Across different Paradigms Starting from Object-oriented to Service-oriented

There are a number of different detection and correction techniques that crossed domains. These approaches mostly focused on source code analysis and evolved from detection to correction in OO and SOA. We extracted data from 78 primary studies and presented the extracted results in an Excel sheet available online[31]. The attributes selected for the extraction for each study is reported in Section3. The analysis results give us a clear idea about the evolution of state-of-the-art approaches in OO and SOA. We divided the research methodology of 78 primary studies in five different categories. As we are interested in OO and SOA, we divided the research techniques reported for OO-related primary studies that also crossed to SOA. The detection and correction approaches used source code metrics or source code analyses as the primary techniques, further combined with other research techniques for the identification of smells. As shown in the following, we use pre- and post-conditions because techniques mainly used behavioral analysis of systems as a pre-requisite [5].

3.4.1 Source Code Metrics

Source code metrics quantify the application features in the OO design knowledge base. These metrics are selected based on the object-oriented design principles. Moreover, these principles are the core of OO design that further classifies the

knowledge based on their definitions and different rules used for these definitions. There should be concrete knowledge about the selection of suitable metrics to check if these metrics are a valid indicator of detected smell or not. However, most of the source code metrics are not applied directly to the source code. A literature review indicates that parsing is the activity that mostly used to get the intermediary representation of source code and then source code metrics are applied to check various quality indicators for the applications [16, 55, 65]. It is also observed that OO source code metrics are used for SO systems to check the quality of the services by detecting several types of defects in services or in their interfaces [49, 51, 53]. Figure 3.9 reports the condition used for source code metrics-based evolution.

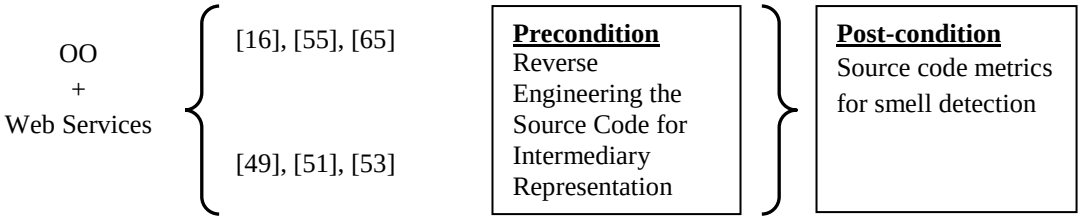


Figure 3.9: Source Code Metrics in OO and SO

The primary studies used source code metrics for OOSE evolve for service-oriented software engineering. All studies used an intermediate representation of source code for the detection of smells across two paradigms.

3.4.2 Mining the Source Code using SVN or CVS

There are a number of studies that report the detection of smells through mining source code using version control systems. Software developers often rely on subversion to keep track of current and historical versions of files like source code, Web pages, and documentation. Software version history is often used to check the relationship between different quality indicators *w.r.t.*the system performance and solution, *i.e.,* refactoring, for a specific problem. These types of studies often use development history of the various releases of the system to check the relationship between two different variables like smells *vs.* maintenance effort [77, 112] or smell *vs.* quality of code after refactoring [104] or smell *vs.* change history information about different versions of the systems [17] by examining the history using SVN or CVS after collecting commits for each change. Approaches also use versioning history with the algorithm called HIST (Historical Information for Smell deTection) [17, 103] and function as follows:

(i) Versioning systems are used to extract changes in source code;

(ii) The locations of the changes from versioning systems are given as input;

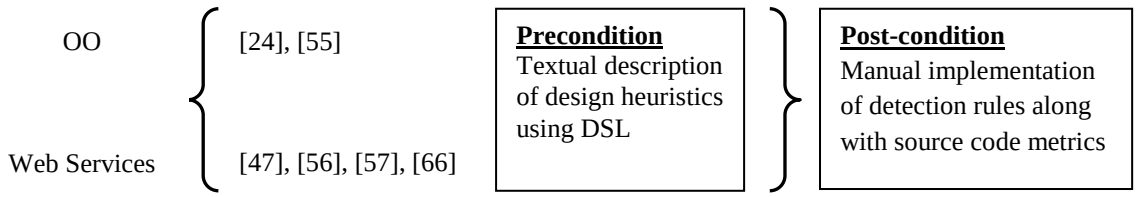
(iii) Change history extractor like SVN or CVS is used to mine the versioning systems, reporting the complete information change. This is performed by comparing the folder and snapshot of change. The srcML[30] toolkit is used to parse the source code to find cases of change. Then, the code smell detector is applied for smells.

3.4.3 Domain Specific Language

Domain analysis is a process that uses specific information required to develop software systems in such a way that is making the desired system reusable for the creation of a new system [116]. The domain-specific languages that are proposed for code, design, and architectural smells are based on following steps:

1. Key characteristics of smells from the literature are gathered and rules to discover them within code or design are designed manually;
2. The next step is to check the measurable properties by using low, high, and medium threshold implementation;
3. Lexical properties using WordNet are examined. Moreover, the properties can be combined using set operators like Union (UNION) and Intersection (INTER) to build more complex detection rules;
4. Classification of the key characteristics is used to divide the properties further;
5. Finally, a DSL (Domain Specific Language) is proposed to describe smells in terms of their measurable, structural, and lexical properties via rule card using a set of operators.

We observed from the systematic literature review of code and design smells that methods based on a DSL mostly rely on the BNF (Backus-Naur-Form) for the specification of smells and the box-plot statistical technique for adjusting the threshold values of source code metrics. This technique evolved from OO [55, 24] to SO [45, 56, 57, 66] as shown in Figure 3.10.



[30]http://www.srcml.org/tools/index.html

Figure 3.10: DSL Evolves in OO and SO

3.4.4 Genetic Algorithm

Smells detection and correction approaches also use genetic algorithms to improve system quality by detecting smells as well as suggesting refactoring opportunities to correct them

[12, 79, 90, 102]. The main benefits of using such approaches are as follows:

1) Genetic algorithms only require defect examples and not different defect types;
2) It is not required to write the detection and correction rules;
3) Metrics with related threshold values are not required that may cause problems in case of different threshold reported in theliterature;
4) The effort required to perform refactoring is also considered for the detection and correction of smells;
5) Already discovered smells results are used as examples;
6) Derived detection rules are used to select best refactoring solutions from a list;
7) Refactoring solutions provide suggestions of the best available option for the correction of defects;
8) Mutation and crossover operators are applied with given probabilities, the resultant is evaluated using a fitness function, and the process is repeated until finally stopping criteria is met;
9) The algorithm, called NSGA-II (Non-dominated Sorting Genetic Algorithm) having the precision for the detection of smells and correction about 87% both for the OO [12, 79, 90] as well as Web services [102]. Figure 3.11 shows the relevant studies based on genetic algorithm.

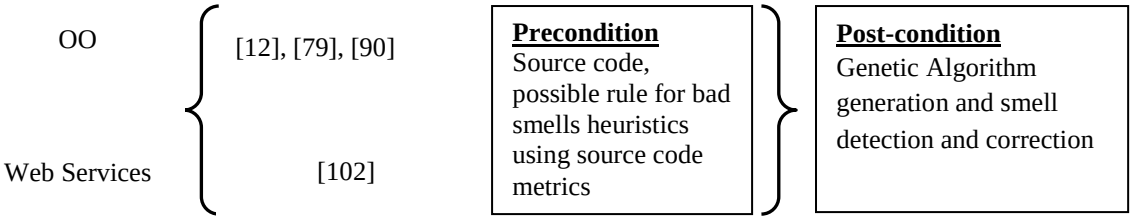


Figure. 3.11: Genetic Algorithm Evolves in OO and SO.

3.4.5 Parallel Evolutionary Algorithm (PE-A)

The Evolutionary Algorithms (EA) and Practical Swarm Optimisation (PSO) process are used to reduce the computational complexity of the search process. The algorithm is based on the following main features:

1) Parallelisation allows speeding up the search process;
2) Exchanging information between different search methods;
3) Using several types of evolutionary algorithms reduces the sensitivity of different parameter used for the detection of smells;
4) Iterations are independent of the problem;
5) The parallelisation process uses a single solution from the search space. The solution uses a set of detection rules that help to detect a specific type of code smell;
6) Parallelisation is used to generate a detection rule, and then the genetic algorithm is used for the detection. Finally, the set of the best candidates as a solution is selected;
7) The PE-A algorithm was reported primarily for OO [70] and then adapted for Web services antipatterns detection [54]. Criteria for implementing PE-A are presented in Figure 3.12.

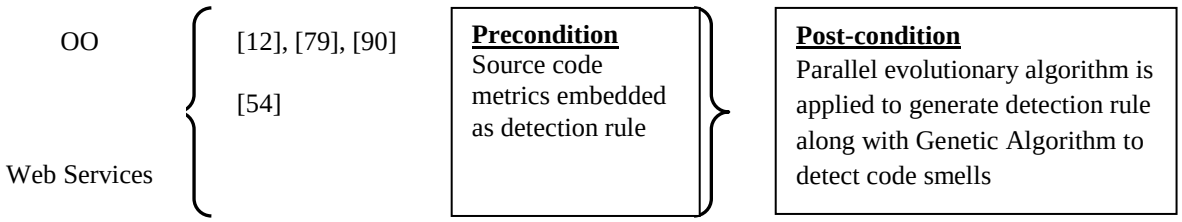


Figure 3.12: PE-A Evolves in OO and SO.

After examining the studies from the last 17 years, we identified five different approaches that evolved in OO and SO. These approaches are now quite matured with the ability to provide highly accurate detection results both for OO and SO paradigms.

We investigated their steps involved in the identification or correction of smells. However, in the literature, the technique related to mining the source code using versioning systems is still not applied for Web services. We did not find any study that discusses the effect of smells across the different versions of service interfaces APIs (Application Programming Interfaces). Source code metrics are the only technique used repeatedly, however, much work still needed to be done for Web services by introducing some novel metrics, which would help to investigate the quality of service issues for Web services.

3.5 Bad Smells that are studied for a specific Paradigm

3.5.1 Smells Reported in OO

A key argument for investigating smells is that certain smells are emphasised more in the literature than others. Moreover, there are different terms provided in the literature for smells like code smells, design smells, architectural smells, and lexical smells that may confuse researchers on which category a smell belongs. The term 'code smell' was first introduced by Fowler [8] with corresponding refactoring opportunities. Later, Brown *et al.* [40] introduced the term 'antipatterns' and divide them into three categories: software development, architectural, and project management antipatterns. Therefore, the smells were later reported in the literature as design, architectural, and code smells, and commonly referred to as *smells*.

We search the relevant literature on bad software smells and identify various smells that are reported as code smells, design smells, architectural smells, and antipatterns. Most of the relevant studies reported and analysed*Feature Envy* as code and design-level smell. To the best of our knowledge, we did not find this smell reported as architectural smells. Table 3.13describes the number of reported smells and their categorisation as bad, code, design, and architectural smells in the literature. In Table 3.13, the Frequency column shows that *Feature Envy* gains an utmost attention from researchers. In contrast, much study still required to be done for the detection of smells like *Yoyo Problem, Un-named Coupling, Extensive Coupling*, and so on, which gained less attention so far in the SE research community.

If we consider the category of smells defined by Fowler [8] and the antipatterns as defined by Brown *et al.* [40], then the total numberof smells comes to 46. If we examine the literature review from the year 2000 until 2017 we find in total 22 smells among the ones defined by Fowler [8]. Also, we were unable to find few smells as

reported by Brown *et al.* [40]. Table 3.14 lists all the smells reported as code, design, architectural smells. However, no studies were found exploring the smells like *Dead End*, *Reinvent the Wheel*, *Primitive Obsession*, *Inappropriate Intimacy*, *Golden Hammer*, and *Incomplete Library Class*. Moreover, there is no template described for code smells as reported for antipatterns in the literature [40]. The correction of code smells might improve the understandability and maintainability of the source code. However, one can remove the antipatterns at the design-level, which may lessen the number of smells at the code-level. Therefore, to improve the system quality, one should remove both antipatterns and code-level smells, which exist at the design and code-level, respectively.

Mantyla [37] and Wake [36] proposed a classification for smells. Moha *et al.* [24] divide the code smells and antipatterns as inter and intra-class smells based on structural, lexical, and measurable properties. Another classification of smells was reported in the literature that divides code smells detection approaches into seven broad categories [70]. However, this categorisation [70] is based on the approaches used to handle smells and not based on the properties of the smells. In this paper, we categorise the smells reported in the literature based on the properties associated with each smell and follow the criteria defined by Mantyla [37]. We collect relevant definition and properties of each smells from the literature and then divide those smells in different classification like code smells, design smells, antipatterns, and architectural smells. Moreover, Mantyla [37] focused only on the code smells defined by Fowler [8], and not on the antipattern properties as defined by Brown *et al.* [40]. Therefore, we also use the classification of design smells and antipatterns reported in [97]. The partition of smells according to the classification reported in the literature is discussed in the following.

***The Bloater*:** Bloater describes something in the source code that has grown rapidly, and, thus, not possible to handle effectively. The smells in this category are*Blob*, *God Class*, *God Method*, *Data Clump*, *Long Method*, *Large Class*, *Primitive Obsession*, *Long Parameter List*, *Complex Class*, and *God Package*. It is very difficult to modify or maintain large codes that further transformed into the *Long Method*, *Large Class*, or *God Class*. This is also true for the *Long Parameter List* and *Data Clump* as they are often found with a long list of parameters. The *God Package* smell is only reported in two studies [65, 75] while the *God Method* is found in three studies [58,

74, 77]. The *Data Clump* smell is also reported in several studies [64, 74, 77, 82, 95, 99, 105]. The smells that are reported by the maximum numbers of studies include *Blob* and *God Class*, 18 and 19 times, respectively. The complete list of references that studied the Bloater group of smells is presented in Table 3.13.

***Object-Oriented Abusers*:** The smells in the object-oriented abuser category include *Switch Statements*, *Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces, Parallel Inheritance Hierarchies and Poor Inheritance Hierarchies, Class Data Should be Private, Global Variables, No Polymorphism, Procedural Class, Public Fields, Missing Association, Cyclic Inheritance, Idle Cut Point, Redundant Cut Point, Traditional Breaker, Adapter, Code Clone, External Duplication, and Cyclic Dependency*. This categorization is often related to the smells where the solution does not fully utilize all the benefits of object-oriented design. The *Refused Bequest* smell is based on this definition because it violates the rule of inheritance design, which is one of the fundamental principles of object-oriented design. Moreover, the *Alternative Classes with Different Interfaces* smell also suffers from the common interface for closely related classes. This shows an example of misusing the object-oriented principles. Similarly, the *Violation of polymorphism*, *Use of Public Data Members*, *Class with Missing andNo Associations* also fall in the abusers category. However, these smells are not reported in the Fowler's catalogue of smells. Several studies under this category of smells ignored *Parent Bequest* [24, 55, 58, 62, 64, 65, 74, 75, 77, 91, 104], whereas the*No Polymorphism* and *Procedural Class* smells were studied in only two studies [59, 80]. Similarly, the *Cyclic Inheritance* smell is reported in only one primary study that clearly shows a research gap for this smell.

***The Encapsulators*:** The encapsulators often deal with the communication mechanism or encapsulation. The smells in this category are *Message Chains*, *Common Method in Sibling Class,* and *Poltergeist.* These types of smells are often inter-dependent where the removal of one smell may cause the introduction of another smell when it itself is removed. The potential solutions for this category of smells are to restructure the class hierarchy by moving a method to another class. However, care must be taken such that the move does not introduce the *Common Method in Sibling Class* smell. The encapsulators smells are mostly based on the way how the object, data, and operations are accessed. The studies that reported the smells in the category

of encapsulators mostly consider *Message Chain* smell [25, 59, 64, 80, 82, 93, 104, 105], the *Middle Man* smell is reported three times in the literature [64, 93, 95], and the *Common Method in Sibling Class* is reported only once [86]. More studies are required in this category.

***The Coupler*:** These types of smells are strongly related to some properties of the class that may hinder the reusability of the software.The *Schizophrenic Class, Message chain, Middle Man, Incomplete Library Class Feature Envy, Inappropriate Intimacy, Intensive Coupling, Extensive Coupling*, and *Un-named Coupling* smells belong to this category. These smells are largely related to the property of coupling, and often misuse or overuse the coupling. The research on smells reported *Feature Envy* as a maximum number of smells detected, corrected, and--or consider for maintenance (*i.e.*, 25 times) as compared to *Intensive*/*Extensive Coupling* [91].

***The Design Rule Abusers*:** These types of smells violate the rules to design the classes or overall programs. These types of the smells are erroneously introduced by the programmers in a way that they might consider them as patterns (*i.e.*, good practice), but later, they turn into antipatterns (*i.e.*, poor practice). Design rule abusers can be further divided as the use of wrong programming approaches like *Boat Anchor, Lava Flow*, or *Wrong Methodology* by using *Copy-Paste Programming, Golden Hammer, Defactoring, Spaghetti Code, Anti-Singleton, Misplaced Class, Wide Subsys Interface, and Yoyo Problem*. The trend towards the smells as defined by Brown *et al.* [40] are not observed much as we found only the *Spaghetti Code* reported in a substantial number of studies [12, 22, 23, 25, 27, 38, 55, 59, 70, 79, 80, 89, 95, 102, 104] as compared to *Yoyo Problem* [71]. However, we are unable to find any relevant study that reports *Boat Anchor* and *Lava Flow* smells.

***The Lexical Abuser*:** Fowler [8] defined code smells in code comments that are smells when the comments do not contain information corresponding to the source code and its behaviour, which Moha *et al.* [24] later reported as lexical smells if they do not match with the internal code behaviour. Recently, a study reports the catalogue of lexical smells based on the internal code structure [15]. This catalogue considers the method, class naming conventions as well as method return types to define lexical smells [15]. The complete list of these smells is available online[31].

31http://research.ciitlahore.edu.pk/Groups/SERC/SOA.aspx

Table 3.13reports the name of the smells along with their category as reported in the corresponding research article. The studies reported in this SLR include 58 smells. However, we do not add *Code Clone* or *Copy Code* to this category as it is not included in our review protocol. In Table 3.13, we report 56 smells along with their frequency.

Table 3.13: Smells Reported in the Literature from the OO Paradigm

Sl.	Smell Name	Type of Smell	Ref. No	Frequency
1	Feature Envy	BS, CS, DS	[17], [24], [58], [62], [64], [65], [70], [74], [75], 77, [81], [83], [84], [85], [90], [91], [93], [94], [97], [98], [99], [103], [104], [105], [106]	25
4	God Class	AS, BS, CS, DS	[16], [24], [58], [60], [62], [65], [71], [74], [75], [77], [81], [85], [91], [97], [98], [101], [105], [106]	19
2	Blob	BS, CS, DS, AS	[12], [17], [22], [23], [25], [27], [38], [55], [59], [70], [76], [79], [80], [89], [95], [102], [103], [104],[119]	18
3	Data Class	CS, DS	[12], [16], [60], [64], [65], [69], [70], [71], [74], [75], 77, [81], [85], [86], [91], [97], [105], [106]	18
5	Long Parameter List	AP, BS, CS, DS	[22], [25], [27], [38], [55], [59], [64], [69], [70], [80], [86], [90], [93], [96], [98], [104], [106],[119]	17
6	Spaghetti Code	AP, AS, CS, DS	[12], [22], [23], [25], [27], [38], [55], [59], [70], [79], [80], [89], [95], [102], [104]	15
7	Shotgun Surgery	CS, DS	[17], [25], [24], [59], [62], [64], [65], [70], [74], [75], [77], [80], [91], [103], [106]	15
8	Duplicated Code	BS, CS, DS	[25], [58], [59], [60], [64], [74], [77], [80], [86], [91], [92], [96], [97], [98], [106]	15
9	Large Class	AP, BS, CS, DS	[22], [25], [27], [38], [55], [64], [69], [83], [86], [90], [92], [95], [96], [98], [106]	15
11	Long Method	AP, BS, CS, DS	[22], [25], [27], [38], [55], [64], [69], [86], [89], [93], [95], [98], [104], [106] ,[119]	15
10	Speculative Generality	AP, BS, CS, DS	[12], [22], [23], [27], [38], [55], [64], [80], [82], [89], [91], [102], [104]	13

12	Lazy Class	AP, BS, CS, DS	[22], [27], [38], [55], [64], [70], [83], [84], [90], [93], [95], [104]	12
13	Refused Parent Bequest	CS, DS	[24], [55], [58], [62], [64], [65], [74], [75], [77], [91], [104] ,[119]	11
14	Functional Decomposition	AS, CS, DS	[12], [22], [23], [25], [55], [59], [70], [79], [80], [89], [102]	11
15	Message Chain	BS, CS, DS	[25], [59], [64], [80], [82], [93], [104], [105], [114],[119]	9
16	Data Clump	BS, CS, DS	[64], [74], [77], [82], [95], [99], [105], [114]	8
17	Swiss Army Knife	AP, CS, DS	[22], [25], [27], [38], [55], [59], [80]	7
18	Divergent Change	CS, C&D, DS	[17], [25], [59], [64], [80], [103]	6
19	Switch Statement	BS, CS	[64], [69], [82], [86], [93], [115]	6
20	Comment	CS, C&D, DS	[25], [59], [64], [80], [95]	5
21	Parallel Inheritance	CS	[17], [64], [69], [103]	4
22	Misplaced Class	CS, DS	[65], [74], [75], [77]	4
23	Class Data Should be Private	AP, CS, DS	[22], [27], [38], [104],[119]	4
24	Poltergeist	AS, CS, DS	[71], [92], [95], [101]	4
25	God Method	CS	[58], [74], [77]	3
26	Anti-Singleton	AP, DS	[22], [27], [38],[119]	3
27	Complex Class	AP, DS	[22], [27], [38],[119]	3
28	Middle Man	BS, CS	[64], [93], [95], [114]	4
29	Brain Class	CS, DS	[81], [85], [91]	3
30	Public Fields	CS	[86], [99]	2
31	Schizophrenic Class	CS	[91], [105]	2
32	God Package	DS	[65], [75]	2
33	Wide Subsystem Interface	DS	[65], [75]	2
34	Decorator	BS, DS	[75], [98]	2
35	Global Variables	C&D, DS	[59], [80]	2

36	No Polymorphism	C&D, DS	[59], [80]	2
37	Procedure Class	C&D, DS	[59], [80]	2
38	Brain Method	CS	[91]	1
39	Common Methods in Sibling Class	CS	[86]	1
40	Extensive Coupling	CS	[91]	1
41	External Duplication	CS	[105]	1
42	Idle Cut Point	CS	[84]	1
43	Intensive Coupling	CS	[91]	1
44	Redundant Cut Point	CS	[84]	1
45	Traditional Breaker	CS	[91]	1
46	Adapter	DS	[74]	1
47	Code Clone	DS	[72]	1
48	Cyclic Inheritance	DS	[92]	1
49	Cyclic Dependency	DS	[72]	1
50	Delegated	DS	[83]	1
51	Interface Bloat	DS	[71]	1
52	Missing Association Class	DS	[92]	1
53	Observer	DS	[71]	1
54	Poor Inheritance Hierarchy	DS	[72]	1
55	Un-named Coupling	DS	[72]	1
56	Yoyo Problem	DS	[71]	1

In Table 3.13, we can observe that the smells reported by Brown *et al.* [40] as antipatterns still did not receive significant attention from the researchers and were not studied thoroughly in the literature. Moreover, a number of new smells are introduced in the literature like *Code Clone*, *Unnamed Coupling*, and *God Package* in addition to the smells defined by Fowler [8].

3.5.2 Smells Reported in SO Systems

(i) Smells that received more attention: Service-Oriented Architecture (SOA) is a promising architectural style that facilitates the development of low-cost, reliable, and flexible services usable or accessible over the Internet [118]. This architectural style can be implemented using technologies like REST, SOAP, SCA, RPC, and J2EE. The detection of smells in the services is quite a new but challenging area that is receiving the increased attention in the SE research community. There are a number of smells that are detected in different SOA technologies like SOAP (*e.g.*, [47], [54]), REST (*e.g.*, [56], [57]), and SCA [*e.g.*, 66]. Moreover, we also found several studies that reported smells for WSDL file that is the core specification of the SOAP Web services. Table 3.14 highlights the frequency for a specific smell reported in the SO systems literature. It is also to be noted that there are more than ten smells reported only once in the services domain like *Breaking Self-descriptiveness*, *Content Negotiation*, and *Ignoring MIME Type* [56]. The REST architectural style is the area where we identify the gap for the detection of various service antipatterns in SO systems.

Table 3.14: Smells that are Reported Repeatedly in the Services Literature

Sl.	Smell Name	Reference No	Frequency
1	**God Object Web Service**	**[24], [47], [54], [62], [65], [66], [67], [68], [75]**	**9**
2	**Low Cohesive Operation**	**[47], [48], [49], [51], [53], [54], [100]**	**7**
3	**Ambiguous Names**	**[48], [49], [51], [52], [53], [100]**	**6**
4	**Chatty Service**	**[47], [52], [54], [66], [67], [68]**	**6**
5	**Data Web Service**	**[47], [52], [54], [66], [67], [68]**	**6**
6	**Duplicated Web Service**	**[47], [54], [66], [67], [68]**	**5**
7	**Enclosed Data Model**	**[48], [49], [51], [53], [100]**	**5**
8	**Redundant Data Model**	**[48], [49], [51], [53], [100]**	**5**
9	**Whatever Types**	**[48], [49], [51], [53], [100]**	**5**
10	**Empty Messages**	**[49], [51], [53], [100]**	**4**
11	**Bloated Service**	**[66], [67], [68]**	**3**
12	**Bottleneck Service**	**[66], [67], [68]**	**3**
13	**Nobody Home**	**[66], [67], [68]**	**3**
14	**Sand Pile**	**[66], [67], [68]**	**3**
15	**Service Chain**	**[66], [67], [68]**	**3**

16	**Stovepipe**	**[66], [67], [68]**	**3**
17	**The Knot**	**[66], [67], [68]**	**3**
18	**CRUDy Interface**	**[47], [54]**	**2**
19	**Fine-grained Web Service**	**[47], [54]**	**2**

(ii) Smells that received less attention: We also found several smells from different SOA technologies that are reported only once in the literature. These smells are detected based on static and dynamic source code analysis by computing different properties and detected instances of each antipattern in the related services. Most of these antipatterns are related to REST services and reported after the year 2014. Table 3.15 and Table 3.16 report the findings of those smells that received less attention from SOA technologies.

Table 3.15: Smells Reported for the First Time in the Services Literature.

Ref	BSD	CN	CvCLRN	DNU	DOR	DSSS	FH	HvNHN	IC	IMT	ISC	ILC	LDFINT	LFDS	LFEBAOD
[48]											√	√			
[50]	√			√	√	√							√	√	√
[52]															
[56]		√					√	√		√	√				
[57]			√					√							

*BSD (Breaking Self Descriptiveness), CN (Content Negotiation), CvCLRN (Contextualised vs. Contextless Resource Name), IMT (Ignoring Mime Type), IC (Ignoring Cache), ISC (Ignoring Self Descriptiveness), FH (Forgetting Hypermedia), HvNHN (Hierarchal vs. Non-Hierarchical Node).

Table 3.16: Smells Reported for the First Time in the Services Literature.

Ref	LFRDD	LSDASI	LFCWSDL	MC	MS	NS	RPT	RC	SINS	SPN	TVA	TTG	TTP	UCFISM	VCU
[48]							√							√	
[50]	√	√	√						√						
[52]					√	√									
[56]				√				√				√	√		

[57]										√	√				√

*TTG (Tunneling Through Get), TTP (Tunneling Through Post), SPN (Singularized vs. Pluralized Nodes), MC (Misusing Cookies).

This research question aboutthe smells (reported for a specific domain) is usefulfor the future research directions. Our findings suggest that most of the study is performed to investigate *God Class, Feature Envy*, *Data class*,and *Blob*. In total researchers discussed 56 smells. Researchers used static or dynamic source code analysis for the identification of smells but attention must be given to those smells that still took less attention like *Un-named Coupling, Poor Inheritance Hierarchy, and Yoyo Problem.* Similarly, there is a need for an investigation on smells that received less attention in theSO paradigm. Most of the smells belong to REST services and still did not receive much attention as they need dynamic analysis of the service interface. The techniques used in the OO for static source code analysis are not applicable to REST services because the method for using a class in OO and the method for consuming services in SO paradigm are not conceptually similar. Most of the attention is given to smells detection for WSDL interface for Web services. The SOAP services are also analysed using dynamic properties like availability, throughput, and response time.

3.6 Correlation between Smells across the Paradigms

The studies in the literature considered different smells that are evolved in OOand other domains of software engineering. These smells are also studied and analysed in the paradigm of SOA (Service-Oriented Architecture). This research question will cover the evolution history of source code measures used for the identification of smells in OO and SO, and the smells evolved in OO and SOA. The OO use classes as compared to services that use interfaces. But services used by the clients are also embedded using classes and methods, thus there is a need to study the evolution of smells in OO and SO. Moreover, service-oriented systems can be implemented by using various technologies and there are different tools, *e.g.*, Java2WSDL, used to generate a representation of services from object-oriented code [48], [49], [50]. SOAP services can be implemented using code-first [48,49] or contract-first approach [78]and OO source code metrics can be reused to detect antipatterns for the code-first approach. If OO code is smelly then the interface generated using an automated tool

will be also smelly [49]. Therefore, source code metrics are highly used by researchers to extract smells from Web services.

Intermediate representations are useful for the identification of smells both for SOA and OO and enable researchers to extract properties of smells. SOA relies on services that generally gather and implement low cohesive operations in comparison to OO where cohesion must be high for a class or sets of methods. LCOM (Lack of Cohesion among Method) may be used for both OO and SOA but the threshold values may vary as reported in previous studies [47], [54].

Evolution of Smells across the Paradigms: Out of the 78 most relevant studies investigated in this paper, we identified four studies that belong to OOparadigm and reported code comments as the smell that evolves to SO and reported by two SO primary studies (*e.g.*, [48, 50]). It is worth mentioning that source code measures used for the identification of smells for SO systems are also reported for OOSE. The primary studies that reported the smell *Comments* across different paradigms are shown in Table 3.17. The detailed descriptions of these smells with the approaches used for their detection are reported online[31]. Code comments are the property used by the developers in both OO and SOA. However, property used for the identification of *COMMENT* may have different thresholds.

Table 3.17: Smells Evolved in OO and SO.

Sl.	Smell Name	OO Ref#	Services Ref#
1	Comments	[25], [59], [64], [80], [95]	[48], [50]

Source Code Metrics Used for Smell Detection in OOand SO: Service-oriented architecture is an emerging and a new challenging area that is gaining increased research attention. The smells reported for the services paradigm initiated to be introduced after the year 2010. It is worth mentioning that the smells reported for this paradigm also rely on source code metrics that are primarily used by OO smells. However, it is noted that smells from SO systems are based on static, dynamic, and linguistic analysis of source code, documentation, and service interfaces, *e.g.*, WSDL files.

Table 3.18: Source Code Metrics used for the Detection of Services Smells.

Sl.	Smell Name	Ref. No	Frequency	Metrics Used
1	God Object Web service	[24], [47], [54], [62], [65], [66], [67], [68], [75]	9	COH, NOD, RT, Av
2	Low Cohesive operation	[47], [48], [49], [51], [53], [54], [100]	7	NOD, ARIO, WMC, LCOM3
3	Ambiguous names	[48], [49], [51], [52], [100],[121]	6	ALS, RGTS, NVMS, NVOS
4	Chatty Service	[47], [52], [54], [66], [67], [68],[121]	6	COH, ANAO, NOD, RT, Av
5	Data Web Service	[47], [52], [54], [66], [67], [68],[121]	6	COH, ANPT, ANAO
6	Duplicated Web Service	[47], [54], [66], [67], [68]	5	ARIP, ARIO
7	Enclosed Data Model	[48], [49], [51], [53], [100]	5	CBO
8	Redundant Data Model	[48], [49], [51], [53], [100]	5	WMC
9	Whatever types	[48], [49], [51], [53], [100]	5	ATC
10	Empty messages	[49], [51], [53], [100]	4	WMC
11	Bloated Service	[66], [67], [68]	3	NOI, NMD, TNP, COH
12	Bottleneck service	[66], [67], [68]	3	CPL, Av, RT
13	Nobody Home	[66], [67], [68]	3	NIR, NMI
14	Sand Pile	[66], [67], [68]	3	NIR, NMI
15	Service Chain	[66], [67], [68]	3	NTMI, Av
16	Stove pipe	[66], [67], [68]	3	NUM, NMD, ANIM
17	The knot	[66], [67], [68]	3	CPL, COH, Av, RT

18	CRUDy Interface	[47], [54]	2	NCO, ANAO, NOD, RT, Av
19	Fine grained Web service	[47], [54]	2	NOD, CPL, COH
20	Chatty Service	[121]	2	COH,CBO

*NOD - Number Of Operations Declared; ANAO - Average Number of Accessor Operations; RT - Response Time; Av - Availability; CPL - Coupling; COH - Cohesion; NCO - Number of CRUDy Operations; ALS - Average Length of Signature; ANP - Average Number of Parameter in Operations; ANPT - Average Number of Primitive Types; ANIO - Average Number of Identical Operations; ARIM - Average Ratio of Identical Message; NIR - Number of Incoming References; NMI - Number of Method Invocation; NOR - Number of Outgoing Reference; NOPT - Number of Port Types; NTMI - Number of Transitive Method Invocation; WMC - Weighted Method Complexity; LCOM3 - Lack of Cohesion Method 3; ATC - Abstract Type Count; NVMS - Number of Verbs in Method Signature; NUM - Number of Utility Method; CBO - Coupling Between Objects.

The metrics used for the identification of 20 smells reported for SOA used source code metrics that have been previously reported for OO. For example, identification of God Object Web Service used cohesion metrics (*e.g.*, LCOM3) and operation identification metric (*e.g.* NOD) that are also used for the identification of smells in OO [30] but the threshold values may vary for SOA in comparison to OO. The LCOM3 value must be high for OO [2], [30] but it should be low for SOA [6], [7].

In answering RQ4 on the smells and source code measures that evolved in OO and SO, we identified many source code metrics that were used for the identification of OO smells but later reused for the identification of smells related to SO systems. More smells could be investigated and detected after examining the complete list of information already available in blogs, Web sites, and books related to SO systems using the source code-level metrics that belong to the widely known CKMJ[32] suite to discover and even define new antipatterns in the SO paradigm. Some studies also used source code metrics as a pre-requisite for smell identification [38, 81, 88, 98] and then different statistical measures are applied to investigate the effect of smells on subject systems. Therefore, we also used these source code measures to investigate the effect of smells on different versions of SO systems, defect prediction and maintenance. We are more interested in investigating the approach used for the smell identification rather than the characteristics of the smells because OO and SOA smells are not directly comparable. For example, for the *Multi Service* antipattern in SO systems and *God Class* antipattern in OO systems, their presence is at different granularity levels. Thus, the detection methodology may be same but conditions used to implement these

32 https://www.dmst.aueb.gr/dds/sw/ckjm/doc/indexw.html

approaches vary like threshold values for measures, implementation of metrics at code-level or at the interface-level, and the presentation of intermediary source code representation [48], [49].

3.7 Trends in Research on Smells from January 2000 to December 2017

Our study collected relevant research studies based on the Kitchenham guidelines [46]. We have observed a clear research trend for the detection and correction techniques of smells. The research started in the year 2000 on the detection of smells in object-oriented systems and slowly moving towards the correction of detected and reported smells.

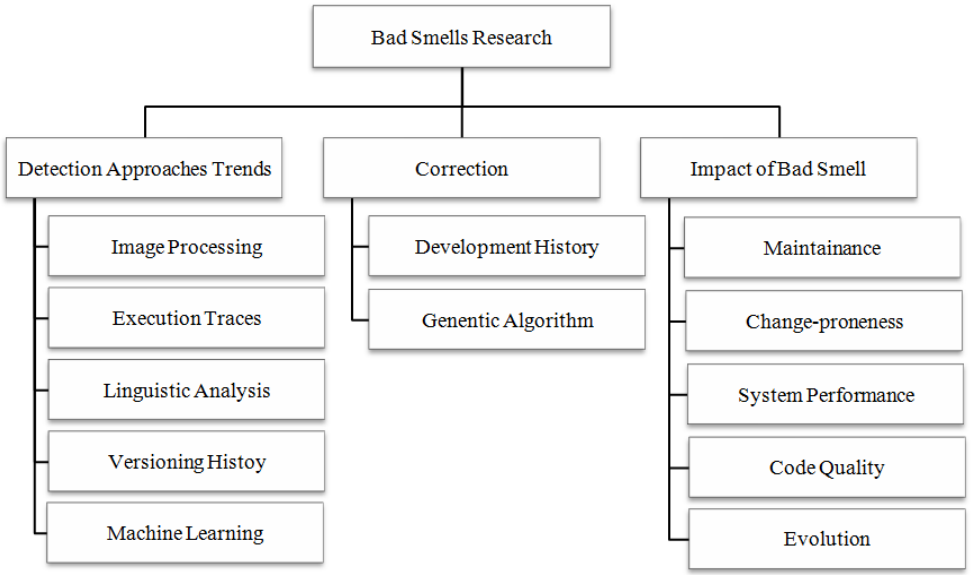


Figure 3.13: General Classification of Research Trend.

Based on our literature review, we can classify research trends on smells into three principal categories, *i.e.*, detection, correction, and the impact of smells. Figure3.13 reports the main research trends in the domain of smells along with their sub-domains. Figure 3.13 provides general trends associated with smells. These research trends are still not reported for SO smells. Figure 3.13 also reports a new trend towards the impact of smells on software system evolution also reported for SO smells [121]. Table 3.19summarises the trend of the research on the smells. Major studies focused on the detection by using source code metrics with the help of different algorithms. Most of the primary studies that use genetic algorithm and development history for the detection of smells also reply mainly on source code metrics.

Table 3.19: Trends in Research from the year 2000 to 2017.

Trend Name	Frequency	Year	Ref#
Smells detection	34	2001-2015	[22], [60], [61], [63], [64], [65], [15], [68], [69], [70], [25], [67], [71], [72], [73], [75], [27], [76], [58], [78], [74], [81], [84], [79], [87], [89], [86], [93], [95], [12], [97], [100]
Smells vs. maintenance	2	2013	[112], [77]
Smells detection using machine learning	2	2011-2012	[92], [26]
Detection of WSDL antipatterns	2	2011-2015	[50-54]
Smells and code quality	2	2012, 2015	[59], [80]
Detection of antipatterns from services	2	2013-2014	[47], [49]
Linguistic antipatterns	2	2015-2017	[59], [64]
Smells impact on software changeability	1	2009	[23]
Smells detection using image processing	1	2010	[88]
Ontological relationship of smells	1	2010	[95]
Antipatterns detection by mining execution traces	1	2013	[24]
Antipatterns and fault-proneness	1	2013	[38]
Impact of smell on system quality	1	2013	[74]
Change-proneness of	1	2014	[66]

service pattern and antipatterns			
Smells correction using the development history	1	2015	[96]
Detection of REST linguistic antipatterns	1	2015	[57]
WSDL refactoring	1	2015	[48]
Performance comparison of smells detection technique	1	2015	[62]
Source code versioning for code smell detection	1	2015	[17]
Evolution of Code smells	1	2017	[121]

Summary on RQ5: In answering RQ5 on the research trends in the domain of smells, we found a corpus of 34 research studies on the detection of smells. We observed (i) the studies on the relationship between smells and the change-proneness, maintenance, and (ii) the studies on inter-smell relationships, both started after the year 2010. A new research gap was filled by introducing smells in SOAP and REST services. Moreover, there is no study published on the correction of REST antipatterns. The natural language processing (NLP) techniques are also used for the detection and correction of smells in OO [16, 104] to the detection of REST linguistic antipatterns [57]. The inter-smell relationships and the impact of smells on the maintenance of services in the SO paradigm have not yet been studied.

3.8 DISCUSSION AND OPEN ISSUES

In total, we reviewed 78 highly relevant studies out of the collection of 506 studies published between January 2000 and December 2017. Previous surveys focused either on only code clones (*i.e.*, a type of smell) or smells or refactoring activities in the OO paradigm. After collecting the most relevant studies related to the evolution of smells in OO and SO, we analysed techniques applied to detect smells. Most of the studies [47, 48, 50, 52, 53, 54, 67] reporting the detection of smells for SO systems are at the interface-level, unlike in the OO paradigm where analyses are mainly done at the source code-level. Moreover, while few studies reported the detection of smells in SO

systems at the architectural-level [54, 55, 56], the correction of these architectural smells is not yet studied and requires further research.

Fowler [8] identified 22 code smells and suggested their refactoring opportunities. Since then, the research on smells had been gaining increased attention, and different research studies were published from the year 2000 until 2017. However, the term 'code smells' later reported in the literature included various forms of code, design, and architectural smells. The development and architectural antipatterns reported in [40] also focus on Fowler code smells. Antipatterns like the *Blob* and *Functional Decomposition* are reported for the first time as antipatterns by Brown *et al.* [40], but later they are also described as code and design smells. Table 3.13 shows the *Blob* as code, design, architectural smells, and antipatterns. Therefore, the software engineering research community does not have a clear consensus on smells. Moreover, the detection approaches for these smells have also varied results due to the availability of multiple source code metrics for the same systems. For example, one can measure the cohesion by using LCOM, LCOM1, LCOM2, or LCOM3 [117].

Moreover, the research community does not always validate their results using precision or recall. We found 15 studies [27, 30, 38, 75, 77, 82, 83, 91, 95, 96, 98, 101, 104, 105, 106] where the precision and recall to measure the accuracy could not be applied because these studies focused on cause-effect relationships, *i.e.*, examples include the impact of smells on the developers' maintenance effort, change-proneness, and so on. One study measured the performance of their research model by using correlation analysis [73]. However, in total, 13 studies did not validate their results at all.

Apache is an open-source FOSS ecosystem providing various versions of Xerces in multiple programming languages, like C++ and Java. More than 90% of the plug-ins reported for the validation of the detection techniques are for Eclipse. We provided our complete findings and detailed discussion on those tools online[31].

While moving towards smells detection in SO systems, most of the studies, *e.g.*, in [49, 51, 53], are dependent on the techniques to develop the SO systems as the code-first or the contract-first approaches. To the best of our knowledge, only one research paper studied smells in RESTful APIs. However, the APIs used for the study are not open-source and the definitions of smells are mostly focused on QoS issues. Future

research should include the impact of smells on different versions of APIs to investigate the evolving smells.

The research on smells is quite mature in terms of a number of studies as well as different state-of-the-art techniques for the OO paradigm. However, the detection of smells in SO systems was introduced only after the year 2010. It is difficult to validate detection or correction techniques of smells for SO systems because open-source service-oriented systems are not greatly available, unlike OO systems. Also, there is no technical support available from the industry to validate the proposed techniques and their results. This research on the impact of smells will also help in confirming the benefits of design patterns in SO systems as opposed to the antipatterns. The area is still open for researchers to study different state-of-the-art smells detection and correction techniques for the improvement of the quality of REST and SOAP systems.

A large number of the studies reported negative impacts of smells on software systems. Few studies investigated the occurrences of smells across different versions of software systems [17, 77, 104, 112]. More investigation should be carried to measure the cause-effect relationship of design patterns *vs.* antipatterns (*i.e.*, smells) as well as their impact on system's overall performance over the long run. The studies focus more on the detection (*i.e.*, 39 studies in total) as compared to the maintenance effort. Some of the studies also use version control system for investigating smells [17]. We did not find any study investigating more than ten smells by using SVN or CVS [12, 58, 60, 103]. Recently, multiple studies have concluded that smells have an adverse impact on software quality [91, 97, 104].

The Implication for Research and Academia

A systematic literature review (SLR) provides directions for researchers who want to understand trends in a specific field. This review on smells provides an updated state of the art on smells starting in the OO and SO paradigms. It also provides guidelines for practitioners working in the software industry to apply smells detection or correction techniques during the software development.

Because the software engineering research community could not find the most appropriate classification of smells, it is essential to develop an automatic oracle for all types of smells, which also includes variants of smells. Studies in the literature also confirm that there is a disagreement on using one definition and implementation of

one specific source code metrics to detect smells. It is a difficult and time-consuming task to manually classify and detect smells. Therefore, we believe that an expert advice is required both from the academia and industry to provide a catalogue on the categories of smells and their detection approaches for those defined by Brown *et al.* [40] and Fowler [8].

For researchers and practitioners, several opportunities are open. An automatic detection technique of smells as part of an IDE must be implemented in Eclipse or Visual Studio for developers and designers to avoid smells. Software development tools must use smells detector to help developers design and implement higher quality systems.

A comprehensive, industrial smells management tool with fully automated detection support with friendly visualisation of the variants of smells as and when they propagate through code segments would help developers. Existing smells detection techniques can be reorganised to have more assistance by the developers for the detection of smells. Origin analysis of smells should also be studied to locate the origin of the introduction of smells in systems across their multiple versions.

3.9 Summary of Literature Review

Smells are classified in the literature as code smells, design smells, architectural smells, and antipatterns. We analysed the most relevant research studies published between January 2000 and December 2017 in different online libraries and investigated five key research questions:

Classifications of the state-of-the-art techniques employed in the detection of smells?

Findings: Mainly two techniques are used in the literature: (1) static source code analysis (*e.g.*, behavioral source code analysis, empirical source code analysis, algorithm-based source code analysis, methodological-based source code analysis, and linguistic source code analysis) and (2) dynamic source code analysis based on dynamic threshold adaptation, *e.g.*, using genetic algorithm, instead of fixed thresholds for smells detection.

State-of-the-art approaches evolve across different paradigms starting from object-oriented to service-oriented

Findings: A number of different detection techniques that crossed domains are based on (1) source code metrics, (2) mining the source code using SVN or CVS, (3) domain-specific language, (4) genetic algorithm, and (5) parallel evolutionary algorithm (PE-A).

Bad smells that are studied for a specific paradigm

Findings: In OO, most of the relevant studies reported and analysed Feature Envy as code and design-level smell. In contrast, much study still required to be done for the detection of smells like Yoyo Problem, Un-named Coupling, Extensive Coupling, and so on, which gained less attention so far in the SE research community. In SO, antipatterns/smells that received most attention include God Object Web Service, Low Cohesive Operation, Ambiguous Names, Chatty Service, and Data Web Service.

Correlation between smells across the paradigms?

Findings: The metrics used for the identification of 20 smells reported for SOA used source code metrics that have also been previously reported for OO. The identification of God Object Web Service used cohesion metrics (*e.g.*, LCOM3) and operation identification metric (*e.g.* NOD) that are also used for the identification of smells in OO. However, the threshold values may vary for SOA in comparison to OO.

Research trends on smells from January 2000 to December 2017.

Findings: We found a corpus of 34 research studies on the detection of smells. A new research gap was filled by introducing smells in SOAP and REST services. Moreover, there is no study published on the correction of REST antipatterns. The natural language processing (NLP) techniques are also used for the detection and correction of smells in OO to the detection of REST linguistic antipatterns.

We identified several issues that should be considered and receive more attention from researchers. We also found several related research activities that must be explored. We advised researchers to pay more attention to linguistic smells that are gaining popularity in the last five years. Moreover, research on the correction of lexical smells requires further investigation. Inter-smells relationship for lexical

smells and performance evaluations of lexical design patterns *vs.* antipatterns are yet to be studied. Refactoring is a major area that is well researched for OO smells but not yet for SCA and REST smells due to their complex nature. Developers' maintenance effort for smells in SO systems is still not addressed in the recent studies. We were also unable to find any antipatterns detected in Java Enterprise systems although their detections were performed on SCA systems [66].

Chapter 4 Specification of Web Services Antipatterns Detection

4.1 Introduction

Design patterns suggest viable solutions to the problems that occur again and again in the design of the software [145]. Design patterns follow the fundamental design principles for the development of software applications. Antipatterns violate fundamental design principles and they are bad solutions to the problems. Antipatterns may have the negative impact on the quality and performance of software applications and their presence may result in degrading the structure of the services [146]. The identification of antipatterns from the web services is a primary step for the removal of antipatterns from service based systems. It is important to have knowledge about the presence of antipatterns in the software system because it will help to improve the software at its abstraction-level. This is reported through different studies that timely detection and correction of antipatterns from software systems improve system performance and quality [113, 149]. This edge motivates researchers to offer assistance for unskilled designers through the detection of antipatterns.

Service Oriented Architecture (SOA) is an arising architecture paradigm that is widely adopted by software industry for the development of distributed and heterogeneous applications. SOA allows the growth of timely, cost effective, flexible, adaptable, reusable, scalable and extendable distributed software applications with enhanced security by composing services through independent, reusable, and platform independent software modules that are easy to get via a network [118]. The application of SOA for emerging technologies such as cloud computing, big data and mobile applications is continuously escalating. Web-services have become a governing technology for Service Oriented Architectures for the development of Service Based Systems (SBS) such as Amazon, Google, eBay, PayPal, Facebook, Dropbox etc. Service based systems needs to evolve with time to fulfill requirements of users. These systems also evolve to accommodate new execution contexts such as addition of new technologies, devices and products [118]. The evolution of service based systems may degrade design and quality of services and it may also cause appearance of common poor solutions called antipatterns. These antipatterns affect the quality of service and can hinder maintenance along with evolution. An antipattern identification techniques revealed that mostly the concentration was on the static analysis of Web-services or on antipatterns in other Service Oriented Architecture technologies (e.g., Service Component Architecture) [66].

It has been reported that antipatterns have impact on the progress and maintenance of software systems [77]. The motivation for automatic identification of SOA related antipatterns is to improve the quality of service and to make maintenance and evolution easy. Maintaining changes in web services is a common practice to provide quality of services to the users. A study has shown that the software maintenance requires eighty to ninety percent of total budget in its whole life cycle [94]. Most state of the art techniques focused on detection of antipatterns from object oriented software applications but these techniques are not capable to detect antipatterns from SOA. We identified only few representative approaches on specification and detection of SOA antipatterns from Web-services [66, 67, 47, 54,15,120,57,56]. To the best of our knowledge, most authors used different metrics and static and dynamic analysis for the detection of antipatterns from web-services. The state of the art antipattern detection approaches has some limitations: The approaches [47, 54] are not extensible for code first and contract first concepts. SODA-W [47] approach detects antipatterns by just considering interface level metrics and it ignores implementation details. PA-E [54] approach detects antipatterns from web services by considering their class as well as implementation details but it is not capable to identify classes that create problem at interface level. Moreover, low cohesion operation and duplicated web service antipatterns are not detected by this approach. The both approaches are also not able to identify location of defected code segments that play a major role for interface level implementation.

The proposed approach is capable to handle code first and contract first concepts. Our approach is free from the implementation restriction of WSDL interface. SOAP services could be implemented by using multiple languages like C#, Java and Perl. Our approach can detect antipatterns from WSDL interface level as well as source code due to support of multiple languages. There are no standard definitions for web-service antipatterns that are important for their accurate detection. Moreover, there is a still no standard benchmark system for comparing and evaluating results of antipattern detection techniques. We present a flexible approach supplemented with a tool support that is used to specify and detect the SOA antipatterns from Web-services. Our objective is to analyze the structure and quality of Web-services and automatically identify antipatterns that may help the progression and growth of Web-services. The proposed approach is implemented by using C# dot.net Framework. We also focus on

improving the accuracy in comparison to existing methods available for detection of Web-service specific antipatterns.

Following are the major contributions of our work:

- Standardized definitions of 10 Web-service related Antipatterns.
- A flexible and scalable approach supplemented with a tool support for detection of 10 Web-service related Antipatterns from the source code and Web service Interface .
- Evaluation and comparison of the proposed approach by performing experiments on different open source Web Services.

4.2 State of the Art for Web services

The number of books are available on SOA-patterns and principles [150,33,132] that provide guidelines and principles characterizing "good" service-oriented designs. These books enable software engineers to manually evaluate the quality of their systems and provide a basis for the enhancement of design and implementation. For example, Rotem et al. [35] suggested 23 SOA-patterns and 4 SOA-antipatterns and described their affects, causes, and corrections. Erl, in his book [33], presented 80+ SOA design, implementation, security, and governance-related patterns. Kr´al etal. [132] elaborated seven different types of SOA-antipatterns, resulted due to the poor practice of SOA rules. Brown et al. [40] provided the set of 40 antipatterns. Dudney et al. [37] presented 52 antipatterns in SOA, and especially in the area of Web-services.

There are few contributions on the identification of patterns from SOA [38,56,39]. Upadhyaya et al. [39] presented an approach to detect 9 SOA patterns. Demange et al. [40] presented an approach to detect five SOA patterns from two SOA based systems.It is revealed through the review of literature that the research on Service Oriented Architecture still needs to be explored. Many detection techniques and tools are presented in the literature [150,24,27,152] that focus on specification and detection of OO antipatterns. These OO based techniques did not give a viable solution for the identification of antipatterns that are pointed out in web-services. There is a difference between structure of Object Oriented software applications and applications developed using web services. A limited number of approaches are available for the identification of the WS antipatterns.

Palma et al. [66] presented a technique supplemented with tool SODA to specify and identify the antipatterns in SCA systems. Authors performed experiments on two different corpora i.e., Home automation system and Frascati service component architecture. Authors apply algorithms that are not generated manually and then they experimented the algorithms on a number of SCA systems to gain the best accuracy. Hence, this approach can only tackle the SCA modules build up using the Java language and are not able to tackle the other SOA technologies like J2EE,SOAP and REST.

Rodriguez et al. [41] described EasySOC and provided a set of guidelines for service providers to avoid bad practices during writing WSDLs. Authors identified eight poor practices that are used to form WSDL template for Web-services. These heuristics are the rules that use pattern matching. A toolset is developed that enforces implementation of guidelines. Authors evaluated effectiveness of the toolset by performing experiments. However, authors have not examined the quality-related issues in the web service design. Coscia et al. [49] have done a statistical correlation analysis on the number of traditional OO metrics and WSDL-level service metrics and found a correlation between them. Antipatterns in SOAP based web services and REST are introduced first time in [37,57,56]. These authors used natural language processing and source code metrics to detect antipatterns. Antipatterns of SOAP based web services are detected with high precision and recall but only for some specific services. The tool SODA-W, an extension of SOFA framework [66] uses already established DSL for the detection of SOAP and REST services.

The state of the art approaches discussed above reflect that a large number of authors worked for the detection of antipatterns from object oriented software projects. We present summarized information about SOA based antipattern detection approaches in Table 4.1. We also realized that SOA based antipattern detection approaches focused towards antipatterns detection for Service oriented architecture specifically for SOAP(Simple Object Access Protocol) based services. We found only three articles for REST APIs antipatterns detection [30,56,57].

Sindhgatta etal. [43] have done a detailed literature survey on service cohesion, coupling, and reusability metrics, and he come up with five new cohesions and coupling metrics, which were set as new service design requirement. Due to

limitations of approaches discussed in the literature, we propose a flexible approach for the specification and identification of SOA-antipatterns in web services.

Table 4.1 :Summarized Information about SOA Antipattern Detection Techniques

Reference	Key Concept	Antipatterns Recovered	Technique Name	Case studies	P /R
[66]	SODA relies on domain specific language that enables antipatterns specification using a set of metrics.	TS, MS, DS,MS	SOFA	Home-Automation System	75%
[67]	SOMAD apply sequential association rules to get execution traces of services .	S,MS,CS,DS,Kt,BS	Association rule mining	Home Automation	90%
[47]	SODA-W is supported by an extended version of SOFA and is dedicated to the specification of SOA antipatterns and their automatic detection in Web services. The extended SOFA provides the means to analyze Web services statically, dynamically, or combining them.	RPT,AN,LCOP,CS,DuS,MRPC,CRUDY-I,GOWS,FGWS	Source Code metrics for static and dynamic analysis along embedded sing DSL	Experiments performed with 13 weather-related and 109 financer elated WSs.	75% 100%
[54]	An automated approach for detection of Web service antipatterns using a cooperative parallel evolutionary	MRPC, CRUDYI, DS, AN,FG,GOWS	Parallel Evolutionary Algorithm	Web services from ten different application domains	85to89 %

	algorithm (P-EA).				
[52]	Genetic Programming approach based on combination of metrics and threshold values	MS,NS,DS,AN	Genetic Programming	310 services of different domains	-85% 87%
[100]	Java to WSDL Mapper	EDM,RPT,WET,AN,UFI,IC,ISM,LCOP	Text Mining and meta programming (Java2wsdl)	60 web services	96% 70-74%
[72]	Contract first concept based approach for detecting WSDL based services using EasySOC tool	WSDL based Services	text mining, machine learning and component based software engineering	391 web services	75-80% 78-94%
[121]	Prediction of Web Services Evolution	Ds,MS,NS,CS	ANN algorithm to predict antipatterns in future release	5 different web services interfaces	81%P,91%R
[43]	Identification of Web Service Refactoring Opportunities as a Multi-Objective Problem	MRPC, CRUDYI, DS, AN,FG,GOWS	MOGP(multi objective genetic programming) to implements detection rule for web services	Different combination of metrics to select dynamic threshold	94%P,92%R

[106]	Comprehensive guidelines along with tool support to enforce these guidelines for service development	EDM,RPT,WET,AN,UFI,IC,ISM,LCOP	WSDL bad practices are embedded in EASY SOC to detect violation of rules in WSDL	Source Code metrics on different web services	95.8%
[53]	Correlation analysis between source code metrics and WSDL implementation code	EDM,RPT,WET,AN,UFI,IC,ISM,LCOP	Effect of source code metrics on WSDl document refactoring	Source Code metrics	NA
[153]	WSDL document improvement for effective service availability	EDM,RPT,WET,AN,UFI,IC,ISM,LCOP	Removlal of antipatterns for service availavility	Results are evaluated using case study using Convertforce WSDl document	NA

P (Precision), NA(Not applicable) ,EDM (Enclosed Data Model), RPT (Redundant Port Type), RDM (Redundant Data Model),WET(What Ever Type) ,AN(Ambiguous names),UFI(Undercover fault Information),IC(Inappropriate Comments),ISM(information within standard messages),LCOP(Low cohesive operations in same port types), MS(Multi Service), NS(Nano Service), DS(Data Service),Kt(the Knot), BNS(Bottle Neck Service) ,CS(Chatty Service),DuS(Duplicated service),SC(Service Chain),NH(Nobody Home), MRPC(may be Its Not RPC).

It is observed that many techniques have used metrics for the identification of the antipatterns.

- Different approaches also used source-code parsing techniques to identify the antipatterns. The parsing techniques used the statistical collection of data like counting Lines of Code, measuring Switch Statement Cases and matching or finding other syntax etc. [49].
- The accuracy of metrics were highly dependent on different values of threshold, and in most cases thresholds were constant [49].

- Accuracy has great impact for the validation. Therefore, many approaches in literature not mentioned the accuracy of the tool like given in [49,100,72].

The light of above analysis, we come up with a new technique supplemented with a tool support: Unification of metrics-based and parsing based approach that not only improves the scope in order to identify number of antipatterns but it also improves accuracy. The required metrics are obtained from the SOAML(Service Oriented Architecture Modeling Language) of Enterprise Architecture, in spite of reinventing the wheel and by examining them directly from the source code.

4.3 SWAD Approach

The specification of web services related antipatterns is primary step for their accurate detection from web services. The specification of antipatterns in literate is textual that is hard to use and describe. Due to unavailability of standard specification of web service antipatterns, we present specification of nine selected antipatterns. Our specification contains detail information that is important to understand and detect these antipatterns. The specifications are further used by our approach for the detection of these antipatterns. We selected these nineantipatterns for the specification and detection due to their common existence in different web services.

4.3.1 Specification of SOAP Antipatterns

4.3.1.1 God Object Antipattern

An object that contains all the information related to the whole service and this object also has many methods. This makes its role in the source-code "god-like".

It is so known as: "Schizophrenic-class", "divergent-change", "unconnected-responsibilities", "conceptualization-abuse", "mixed-abstractions" [37].

Variants: "Vague-classes", "abusive-conceptualization", "non-related data and behavior", "irrelevant-methods", "discordant-attributes".

God object exists if the service contains:

Many Methods AND Has Very Low Cohesion AND Has High Response-Time AND Low-Availability. Where Many Methods >=10, Cohesion >= 1, High Response-Time >=1

4.3.1.2 Data Web-service Antipattern

A web-service that performs information retrieval tasks in a distributed environment through accessor operations like getters and setters.

This Antipattern occurs when a class is used as a holder for data, without any methods operating on it.Data Web service exists if the service contains: High Accessor Operations with Few Parameters And Has High Cohesion And High Primitive Parameters

Where Accessor Operations> 50 < 73 And with Few Parameters And lcom3<=0 And Primitive Parameters >100

4.3.1.3 Fined Grained WS Antipattern

Fine-grained web-service description regards tiny services out of which the larger ones are composed. That larger one needs to have many coupled web-services. Therefore, it gives rise to higher development complexity, reduced usability. Individual Web-service is less cohesive due to related operations that spread across services of an abstraction. This antipattern is the result of overdone implementation-complexity.

The existence of this antipattern based on : Few Operations And Has Low Cohesion And Has Very High Coupling Where Operations >=1 And <=2, Low Cohesion >=1 And <=2, Coupling >=1 And <=4

*4.3.1.*4 Ambiguous Name WS Antipattern

The developers use the key-terms like Port-Types, operation, and message that contains too short and long, or too general terms, or even show the improper use of verbs.

This antipattern arises when the class name has a verb only and hold one method with same name as the class and class has no inheritance. Ambiguous WS antipatterns also known as “Operation-class”, “method turned into class”, “single-routine-classes”.

A service contains: Too Long or Too Short Signatures AND Has Too Many General Terms in Operations ,

where COUNT [Operations signature length<3 or Operations Signature length > 30] >1 or Ambiguous operations name should have any one (arg, var, obj, foo, param, in, out, str) >1

4.3.1.5 Duplicated Service

Duplicated Web-service contains identical-operations with the similar names and message parameters. This aantipattern introduces when two or more abstractions are identical sharing commonalities with their improper use in the design.

A web-service having Identical Operations and Identical Port-Types

Where ARIP > 1 And ARIO>1

ARIP= Average Ratio of Identical Port-Types, ARIO= Avg. Ratio of Identical Operations

Where ARIP count all ambiguous names starting from (arg, var,obj,foo) and ARIO is calculated as all meaningless operations name having length less then 3 and greater then 30. We also used Wordnet dictionary to check the ambiguious names.

4.3.1.6 LowCohesiveOperations in the Same Port-Type Antipattern

Many unrelated operations in one port type.

The service contains: Many Methods And has Very Low Cohesion.

NOD >=1 And <= 70 And ARAO <=27

ARAO= Avg. Ratio of Accessor Operations

4.3.1.7 Redundant Port-Types Antipattern

A WS contains multitude Port-Types and is composed of number of redundant operations handling the same messages. This antipattern arises when two or more classes have split-identity.

Web-service contains: Many Operations AND Has Many Port-Types resulting in High Cohesion

Where NOPT>1 And NPT >1

NOPT = Num of Operations in Port-Types, NPT= Number of Port-types

4.3.1.8 Chatty Web-service Antipattern

Chatty-WS is an antipattern in which numerous attribute-level operations like getters and setters exist in order to complete an abstraction. Difficult to infer the order of invocation give rise to maintenance issues. Chatty Web-services exists if service contains:

Low Cohesion with High Accessor Methods And Has Low Availability And High Response Time And Many Methods.

Where Low Cohesion >0 And Accessor Methods >=101 And Many Methods >70

4.3.1.9 CRUDy Web-service Antipattern

A web-service design that contains CRUD-type operations, e.g., create (), ready (), delete (), update (). Interfaces designed may have several methods need to be called to accomplish a goal which makes it chatty.

A web-service is Chatty if it contains: Many CRUD-type operations and LOW Cohesion AND High Accessor Operations AND high Procedures

LCOM3<=0 AND Accessor operations>100 and procedures >70 Crudy Operations>1

Where CRUD-type operations >1

4.3.1.10 Loosey Goosey Web Service Antipatterns

Services are designed in a complex way that create problems for further service extensibility and functionality. Services are tightly coupled and not able to answer the user request.A web service is Loosey goosey if the service interface implementation is single tier and services are loosely coupled, less cohesive

DIT<1 AND AND CBO>1 AND LCOM3>0

Where DIT: Depth of inheritance

4.3.2 Detection Approach for SOAP Antipatterns

The motivation of our proposed approach stems from the methodology presented by [10] for the detection of antipatterns from Web-services. We input standard definitions of web services presented in the previous section. The definitions include static and dynamic properties of web services. The static properties include static features of web services such as number of operations, number of port types, number of parameters etc. The dynamic properties include features such as response time and availability. The metric rules are composed based on the static and dynamic properties of web services. The approach applies these metric rules for the detection of a specific antipattern.

4.3.2.1 Antipatterns Detection using Reverse Engineering

Step1

Web services interfaces are reversed in form of java code using JAVA2WSDL tool using contract first approach.

Step 2

Antipattern detection engine is build using C# after implementing the static and dynamic properties of each antipatterns collected from step 2. The code-first approach is based on Sparx System data model that is directly generated from source code.Sparx System Enterprise Architect has the ability to generate data model of different languages directly from the source code. Instead of reinventing the wheelSparx System Enterprise Architect has the ability to generate data model of different languages directly from the source code. Instead of reinventing the wheel, we relied on the use of metrics i.e. SOA data model to extract relevant features related to any given antipattern. This approach is similar to the contract-first where we have a WSDL interface and then use tool to generate java code . However, previously reported techniques in literature used OOSE metric to check the antipatterns from SOAP service, no interface level properties are used.

4.3.2.2 SOAP antipatterns detection at interface level

This technique uses WSDL instead of reverse engineering the code in multiple form of representation like C#,C++and java . SOAP web services are analyzed at interface level and their dynamic properties like availability is measured using SAAJ tool .

Step 1:

WSDL code is used as a contract. SOAML is used as WSDL intermediate representation using Sparx system Enterprise architecture.

Step2 :

Antipattern detection engine is build using C# after implementing the static and dynamic properties of each antipatterns collected from step 1. Instead of reinventing the wheel, we relied on the use of Interface level and code level metrics and use SOAML model for the extraction of Interface level properties. The proposed approach adapt variable threshold using GUI that helps analyst to check the antipatterns as per its requirement.

We describe relevant properties of antipatterns in web-services and parse the WSDL code into Java template using EA tool.

1. EA Data Model
2. Structure Query Language

EA Data Model

Our approach is based on Sparx System data model that is directly generated from source code. Sparx System Enterprise Architect has the ability to generate data model of different languages directly from the source code. Instead of reinventing the wheel, we relied on the use of metrics i.e. SOA data model to extract relevant features related to any given antipattern.

We used SOAP-UI for parsing the code and SQL, to extract the required data for the detection of Web-service related antipatterns from the Service Oriented Modeling Framework of Enterprise Architect.We selected Enterprise Architect because of its ability to generate well structured, self-explanatory and detailed (metrics) data model from the source code of 13 programming languages.

B) Structure Query Language

Our approach is based on SQL to extract data from the data model of Sparx System Enterprise Architect. Structure Query Language is very useful database Query language capable to extract any required data from the Database model.

SQL is having enough types and clauses through which we can extract (delete or alter) any required data (if present) from the SQL database. SQL Queries are useful to retrieve huge amount of data and records from database effectively and efficiently. SQL based databases established standards that is adopted by ANSI & ISO. SQL commands syntax is simple English like statements.

Examples of SQL commands that we used in our prototype for extracting data from the data model of Enterprise Architecture are given below:

a) *Portnames = select Name from t_object where t_object.stereotype='WSDLportType'*

b) *Operations =Select t_operation.Name from t_operation,t_object where t_operation.Object_ID=t_object.Object_ID and t_object.Stereotype='WSDLportType'*

c) *Procedures = Select count(operationid) from t_operation,t_object where t_operation.Object_ID=t_object.Object_ID and t_object.Object_Type='class'*

d) *Variables = "select count(*) from t_attribute,t_object where t_attribute.object_id=t_object.object_id and t_object.object_type='class'"*

e) *Methods accessed parameter = select count (name) from t_operationparams ");*

f) *Paramatertype = "select count(operationid) from t_operationparams ");*

g) *Accessoroperation = "select count (name) from t_operation where name like'set*' or name like 'get*'*

h) *Crudy Operation =select count (t_operation.OperationID) from t_operation, t_object where t_operation.Object_ID = t_object.Object_ID AND t_object.Object_Type = 'Interface' and t_object.Name like 'Create*' and t_object.Name like 'update*' and t_object.Name like primitivetype = select count(operationid) from t_operationparams where type IN('boolean','double','int','byte','short','long','char')"delete*'*

i) *Methodsacessed = "select count (operationid) from t_operation ,t_object where t_object.Scope='public' and t_object.Object_Type='class'"*

j) *coupling = select count (connector_id) from t_connector,t_object where t_connector.Connector_ID= t_object.Object_ID and t_object.Object_Type='class'*

The above mentioned SQL queries are used to calculate the Interface level metrics listed in Table 4.2. SWAD used SOAML model for representation of SOAP web services, so the mapping of different features of SOAP along with SOAML are also listed below in Table 4.2.

Table 4.2 : Mapping of SOAP interface with SOAML Model

Sr.No	SOAP Parameter	SOAML	SOAML Table Name	Property map
1	Wsdl :port-Types	Service End point interface	T_operation	'Stereotype'
2	Wsdl:operation	Method,Operation	T_operation	'name'
3	Wsdl:binding	Stub	T_object	'name'
4	Wsdl:service	Service interface	T_object	'type'
5	Wsdl:port	Port Accessor	T_operation, t_operationparams	'Sterotype'
6	Xsd:simple type	Class	T_object	'type'
7	XSD Element and Attribute	Class	T_attribute	'type'

Table 4.2 helps to map the property of SOAML with SOAP services. SOAML parse the SOAP services and map each SOAP property with its corresponding table and we use the attribute of each table to extract the values of the SOAP services. Property map column describes the SOAML database table attribute that helps to extract the value of SOAP attribute like 'port types ' and operations.

C)Prototype Tool

The proposed architecture of the SWAD is described in Figure 4.1 and interface is available in Figure 4.2

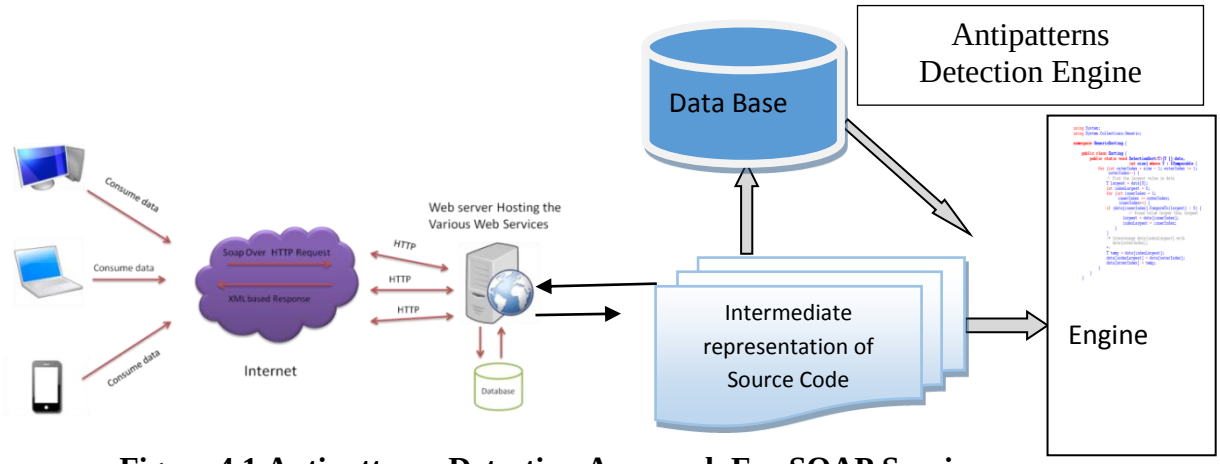


Figure 4.1 Antipatterns Detection Approach For SOAP Services

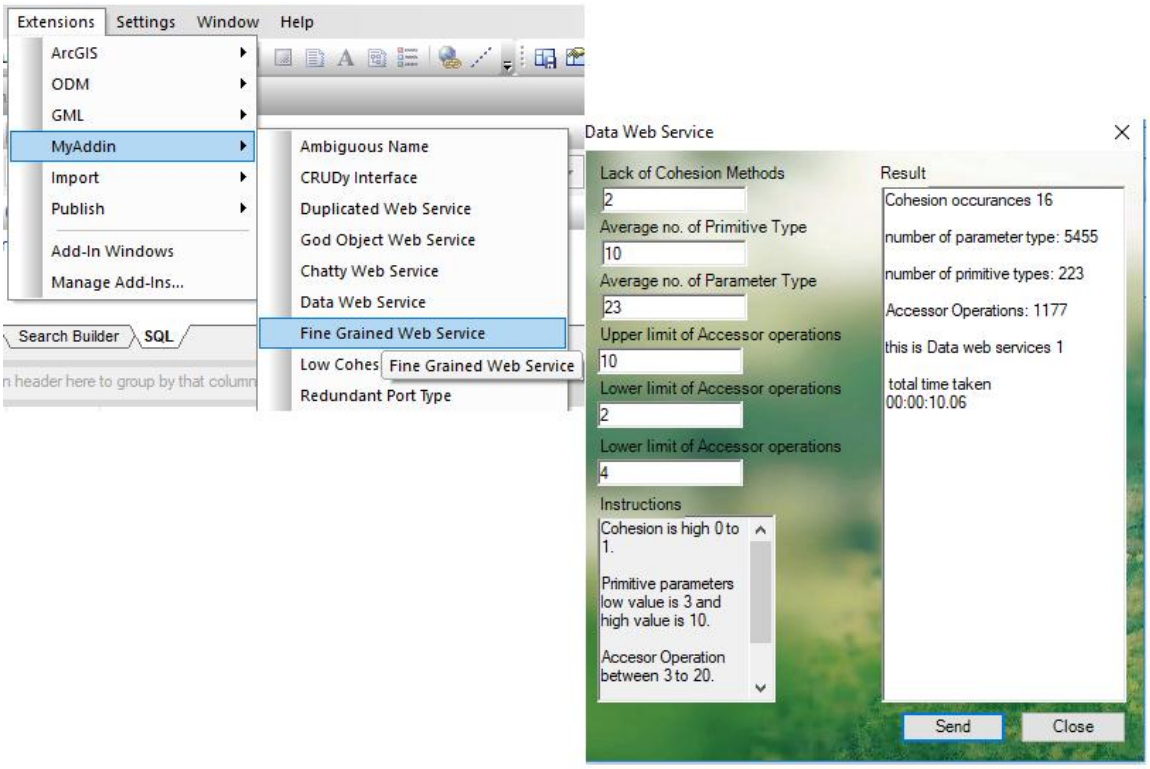


Figure 4.2 Interface of Prototype Tool

4.4 Experimental Results

We selected 60 weather and 7 finance related web services to evaluate our approach and recovered 9 antipatterns. We selected these datasets due to their free availability and comparison of our results with state of the art approaches. The general information of the services under analysis using code-first approach is as reported in Table 4.3

Table 4.3 : Statistics of Examined Systems

Sr.#	Systems	SLOC	Methods	Attributes
1	BLiquidity	12210	4618	4284
2	Cloan to Currency	29663	8647	7650
3	sxBATS	13068	4994	4584
4	xBondRealTime	26577	6170	4541
5	Curs	12415	4627	4333
6	Data	34836	10451	8528
7	ExchangeRates	13535	5030	4544
8	MFundService	13530	4930	4527
9	getImage	16307	5506	5230
10	Index	11958	4635	4218
11	Populate	11335	4406	4077
12	ProhibitedInvestor	11565	4453	4165
13	StockQuoteService	13331	5383	4923
14	StockQuotes	19790	6327	5662
15	sflXML	14941	5771	5079
16	TaarifCustoms	19678	6565	5919
17	TaxEconomy	16167	6210	5336
18	TipoCombio	13268	4959	4608
19	VerifilterSoap	10500	4204	3878

20	WebService	11120	4314	4046
21	wsIndicator	10329	4203	3864
22	wsStrikon	15078	5588	5217
23	xCalender	22122	7294	6585
24	xCharts	32925	5585	3679
25	xCompensation	18917	6553	5693
26	xEarningCalender	20340	7137	6374
27	xEnergy	49670	13952	11433
28	xEnchanges	21305	7154	6476
29	xFinance	49377	14458	11503
30	xFundamentals	23731	7581	6806
31	xFundata	34821	11384	9405
32	xFunds	31660	9765	8193
33	xFuture	58311	1732	766
34	xGlobalBond	16582	5723	5079
35	xGlobalFundamentals	21858	6963	6236
36	xGlobalHistorical	36392	11192	9626
37	xGlobalRealTime	13829	5273	4741
38	xIndices	22838	6775	5978
39	xInsider	35561	11714	9565
40	xInterbank	77971	7551	4291

41	xLogos	11385	4611	4257
42	xMaster	23182	7922	7094
43	xMetals	57469	16208	12659
44	xNASDAQ	21183	7059	5846
45	xNews	15531	5666	5158
46	xOFAC	16037	5906	5293
47	xOptions	24044	7896	6520
48	xOutlook	14899	5353	4937
49	xReleases	4872	5984	5355

The results of our approach are shown in Tables 4.4 and 4.5. Each Table presents the names of web-services in second column and then rest of the columns shows antipatterns with their possible metrics detected.

4.4.1 Results of Antipattern Detection using Code –first Approach

We used two different types of data sets from the field of financial applications and the web services used for weather forecasting. The detailed results are reported in Table 4.4 and Table 4.5.

Table 4.4 Results for Finance related Web-services

Sr.No	Name ofWeb-services	GOWS	DWS	CWS	LCWS	FGWS	CRUDI	RPT	Dup-WS	ANWS	LGWS	RT
1	BLiquidity	√	√	√	√	√	X	X	√	√	√	1s
2	Cloan to Currency	√	√	√	√	√	X	X	√	√	√	2s
3	xBATS	√	√	√	X	X	X	X	√	√	X	2s
4	xBondRealTime		√	√	√	√	X	X	√	√	X	None
5	Curs	√	√	√	√	√	X	X	√	√	X	2s

6	Data	√	√	√	√	√	X	X	√	√	X	2s
7	ExchangeRates	√	√	√	√	√	X	X	√	√	√	2s
8	MFundService	√	√	√	√	√	X	X	√	√	√	2s
9	getImage	√	√	√			X	X	√	√	√	
10	Index	√	√	√	√	√	X	X	√	√	√	2s
11	Populate	√	√	√	√	√	X	X	√	√	√	3s
12	ProhibitedInvestor	√	√	√	√	√	X	X	√	√	√	2s
13	StockQuoteService	√	√	√	√	√	X	X	√	√	X	3s
14	StockQuotes	√	√	√	√	√	X	X	√	√	X	2s
15	sflXML	√	√	√	√	√	X	X	√	√	√	2s
16	TaarifCustoms	√	√	√	√	√	X	X	√	√	√	4s
17	TaxEconomy	√	√	√	√	√	X	X	√	√	X	2s
18	TipoCombio	√	√	√	√	√	X	X	√	√	X	2s
19	VerifilterSoap	√	√	√	√	√	X	X		√	X	2s
20	WebService	√	√	√	√	√	X	X	√	√	X	2s
21	wsIndicator	√	√	√	√	√	X	X	√	√	√	3s
22	wsStrikon	√	√	√	√	√	X	X	√	√	√	2s
23	xCalender	√	√	√	√	√	X	X	√	√	√	2s
24	xCharts		√	√	√	√	X	X	√	√	√	2s
252	xCompensation	√	√	√	√	√	X	X	√	√	√	2s
26	xEarningCalender	√	√	√	√	√	X	X	√	√	√	2s
27	xEnergy	√	√	√	√	√	X	X	√	√	X	2s
28	xEnchanges	√	√	√	√	√	X	X	√	√	X	2s
29	xFinance	√	√	√	√	√	X	X	√	√	X	2s
30	xFundamentals	√	√	√	√	√	X	X	√	√	X	2s
31	xFundata	√	√	√	√	√	X	X	√	√	√	Non

												e
32	xFunds	√	√	√	√	√	X	X	√	√	X	2s
33	xFuture		√	√	√	√	X	X	√	√	X	2s
34	xGlobalBond	√	√	√	√	√	X	X	√	√	√	2s
35	xGlobalFundaments	√	√	√	√	√	X	X	√	√	√	2s
	als											
36	xGlobalHistorical	√	√	√	√	√	X	X	√	√	√	2s
37	xGlobalRealTime	√	√	√	√	√	X	X	√	√	√	2s
38	xIndices		√	√	√	√	X	X	√	√	√	Non
												e
39	xInsider	√	√	√	√	√	X	X	√	√	X	2s
40	xInterbank		√	√	√	√	X	X	√	√	X	2s
41	xLogos	√	√	√	√	√	X	X		√	X	2s
42	xMaster	√	√	√	√	√	X	X	√	√	√	2s
43	xMetals	√	√	√	√	√	X	X	√	√	√	2s
44	xNASDAQ	√	√	√	√	√	X	X	√	√	√	2s
45	xNews	√	√	√	√	√	X	X	√	√	√	2s
46	xOFAC	√	√	√	√	√	X	X	√	√	√	2s
47	xOptions	√	√	√	√	√	X	X	√	√	√	Non
												e
48	xOutlook	√	√	√	√	√	X	X	√	√	√	2s
49	xReleases	√	√	√	√	√	X	X	√	√	√	2s

MO = Multi-Operation Occurrences, CO = Cohesion Occurrences, NPT = Number of Parameter Type, NOD = Number of Operations Declared, AO = Accessor Operations, NOI = Number of Instances detected, DT = Detection Time, RT = Response Time, P = Precision ,R = Recall ,NAN= Num. of Ambiguous names in Port-type, SLAP ,AmbOp =Ambiguous Operations ,ANA=Ambiguous names antipattern,X = No Response (Service not available)

Table 4.5 Results for Weather-related Web-services

Name ofWeb-	**GOW**	**DW**	**CW**	**LCW**	**FGW**	**CRU**	**RP**	**Du**	**ANW**	**LGW**
								p-		

services	S	S	S	S	S	DI	T	WS	S	S
AIP3	√	√	√	√	√	√	X	√	√	√
FindingService	X	X	X	√	X	√	X	X	√	√
Ndfd	√	√	√	√	√	√	X	√	√	X
SOAP WS	X	X	X	√	X	X	X	X	√	X
WeatherForecastService	√	√	√	√	√	X	X	√	√	X
WeatherTerrapin	√	√	√	√	√	X	X	√	√	X
webSky	√	√	√	√	√	X	X	√	√	X

GOWS: Gob Object Web Service , DWS: Data Web Service , CWS: Cruddy Web Service , LCWS: Low Cohesive Web service , RPT: Redundant Port Type , ANWS: Ambiguous Name Web Service , FGWS: Fine Grained Web service, CRUDY I: Crudy Interface, DupWS:Duplicate Web Service

4.4.2Results of Antipattern detectionat interface level using Contract first Approach

Table 4.6 reports the results of SWAD at interface level using SOAML(Service Oriented Architecture Modeling Analysis) framework for the detailed analysis of antipatterns for SOAP web services. This approach is suitable for ***code-first*** technique too.

Table 4.6 Results for Finance related Web-services

Sr. #	Name of Web-services	GOWS	DWS	CWS	LC WS	FGWS	CRUDI	RPT	Dup-WS	ANWS	LGWS	RT
1	BLiquidity	√	√	√	√	√	X	X	√	√	√	1s
2	Cloan to Currency	√	√	√	√	√	X	√	√	√	√	2s
3	xBATS	√	√	√	X	X	X	X	√	√	X	2s
4	xBondRealTime		√	√	√	√	X	X	√	√	X	None
5	Curs	√	√	√	√	√	X	√	√	√	X	2s
6	Data	√	√	√	√	√	X	√	√	√	X	2s
7	ExchangeRates	√	X	X	X	√	X	√	√	√	√	2s

8	MFundServic e	√	√	√	√	√	X	√	√	√	√	2s
9	getImage	X	√	X	X	X	X	√	√	√	√	
10	Index	√	√	√	√	√	X	√	√	√	√	2s
11	Populate	√	√	√	√	√	X	√	√	√	√	3s
12	ProhibitedInv estor	√	√	√	√	√	X	√	√	√	√	2s
13	StockQuoteS ervice	√	X	√	X	√	X	√	√	√	X	3s
14	StockQuotes	√	√	√	√	√	X	√	√	√	X	2s
15	sflXML	√	√	√	√	√	X	√	√	√	√	2s
16	TaarifCustom s	√	√	√	√	√	X	√	√	√	√	4s
17	TaxEconomy	√	√	√	√	√	X	√	√	√	X	2s
18	TipoCombio	X	X	√	X	√	X	√	√	√	X	2s
19	VerifilterSoa p	√	√	√	√	√	X	√		√	X	2s

20	WebService	X	√	√	√	√	X	√	√	√	X	2s
21	wsIndicator	√	√	√	√	√	X	√	√	√	√	3s
22	wsStrikon	√	√	√	√	√	X	√	√	√	√	2s
23	xCalender	√	√	√	√	√	X		√	√	√	2s
24	xCharts	X	√	√	√	√	X	√	√	√	√	2s
25	xCompensation	√	√	√	√	√	X	√	√	√	√	2s
26	xEarningCalender	√	√	√	√	√	X	√	√	√	√	2s
27	xEnergy	√	√	√	√	√	X	X	√	√	X	2s
28	xEnchanges	√	√	√	√	√	X	X	√	√	X	2s
29	xFinance	√	√	√	√	√	X	X	√	√	X	2s
30	xFundamentals	√	√	√	√	√	X	X	√	√	X	2s
31	xFundata	X	√	√	√	√	X	X	√	√	√	None

32	xFunds	√	√	√	√	√	X	X	√	√	X	2s
33	xFuture	X	√	√	√	√	X	X	√	√	X	2s
34	xGlobalBond	√	√	√	√	√	X	X	√	√	√	2s
35	xGlobalFundamentals	√	√	√	√	√	X	X	√	√	√	2s
36	xGlobalHistorical	√	√	√	√	√	X	X	√	√	√	2s
37	xGlobalRealTime	√	√	√	√	√	X	X	√	√	√	2s
38	xIndices		√	√	√	√	X	X	√	√	√	None
39	xInsider	√	√	√	√	√	X	X	√	√	X	2s
40	xInterbank		√	√	√	√	X	X	√	√	X	2s
41	xLogos	√	√	√	√	√	X	X		√	X	2s
42	xMaster	√	√	√	√	√	X	X	√	√	√	2s
43	xMetals	√	√	√	√	√	X	X	√	√	√	2s

44	xNASDAQ	√	√	√	√	√	X	X	√	√	√	2s
45	xNews	√	√	√	√	√	X	X	√	√	√	2s
46	xOFAC	√	√	√	√	√	X	X	√	√	√	2s
47	xOptions	√	√	√	√	√	X	X	√	√	√	None
48	xOutlook	√	√	√	√	√	X	X	√	√	√	2s
49	xReleases	√	√	√	√	√	X	X	√	√	√	2s

MO = Multi-Operation Occurrences, CO = Cohesion Occurrences , NPT = Number of Parameter Type, NOD = Number of Operations Declared, AO = Accessor Operations, NOI = Number of Instances detected, DT = Detection Time, RT = Response Time, P = Precision ,R = Recall ,NAN= Num. of Ambiguous names in Port-type, SLAP ,AmbOp =Ambiguous Operations ,ANA=Ambiguous names antipattern,X = No Response (Service not available)

Table 4.7 Results for Weather-related Web-services

Name of Web-services	**GOWS**	**DWS**	**CWS**	**LC WS**	**FGWS**	**CRUDI**	**RPT**	**Dup-WS**	**ANWS**	**LGWS**
AIP3	√	√	√	√	√	√	X	√	√	√
FindingService	X	X	X	√	X	√	X	X	√	√
ndfd	√	√	√	√	√	√	X	√	√	X
soapWS	X	X	X	√	X	X	X	X	√	X
WeatherForecastSer	√	√	√	√	√	X	X	√	√	X

vice										
WeatherTerrapin	√	√	√	√	√	X	X	√	√	X
webSky	√	√	√	√	√	X	X	√	√	X

4.4.3 Comparison of Results

For the comparison of different detection tools for antipatterns in web-services, we select SODA-W, and Parallel Evolutionary Algorithm the brief description is listed in Table 4.8. The reason of their selection is that we have very limited number of approaches available for the identification of web-services related antipatterns. It is worth mentioning that we report the accuracy of our tool using Code-first(WSDL to java) and contract first (Interface only) approach. We use SOAML(Service Oriented Architecture Modeling Language) to model the system using Sparx system Enterprise Architecture.

Table 4.8 Description of Detection Tools

Detection Tools	**Existing Java Open Source Tools Description**
SODA-W	SODA-W is an approach used to detect antipatterns for SOAP based web services . [47]. (http://sofa.uqam.ca/soda-w//)
P.E Algo	'Parallel Evolutionary Algorithm is based on auto-mated detection algorithms that automatically select threshold for source code metrics used to detect antipatterns for SOAP based services. The tool is not publically available [54].

Table 4.9 Comparison of Results for Weather Related Services

SWAD Tool			**SODA-W Tool**		**PE-A Algorithm**	
Antipattern	***WS***	***Precision (Average of Both Approaches)***	***WS***	***P***	***WS***	***P***

GWS	Detected	78%	None detected	----	Detected	87%
DWS	Detected	100%	None detected	----	Detected	88%
Chatty WS	Detected	68%	Detected	50%	Detected	81%
LCWS	Detected	95%	Detected	100%	None Detected	--
FGWS	Detected	98%	Detected	100%	Detected	83%
DWS	Detected	86%	None detected	----	Detected	90%
ANWS	Detected	93%	Detected	100%	Detected	95%
CRUDy I	None detected	----	Detected	50%	Detected	88%
RPT	Detected	85%	Detected	100%	Detected	87%
MRPC	None detected	----	None detected	----	None Detected	--

Table 4.10 Comparison of Results for Finance Related Services

SWAD Tool			**SODA-W Tool**		
Antipattern	**WS**	**Precision**	**Antipattern**	**WS**	**Precision**
GWS	Detected	42.8%	GWS	None detected	----
DWS	Detected	100%	DWS	None detected	----
Chatty WS	Detected	42.8%	Chatty WS	None detected	----
LCWS	Detected	71.5%	LCWS	Detected	100%
FGWS	Detected	100%	FGWS	Detected	66.67%
DWS	Detected	57.1%	DWS	None detected	----
ANWS	Detected	100%	ANWS	Detected	100%

CRUDy I	None detected	----	CRUDy I	None detected	----
RPT	None detected	75%	RPT	Detected	100%
MRPC	None detected	----	MRPC	None detected	----

4.4.4 Comparison of Results with P.E Algorithm

Figure 4.3 shows the comparison of the Antipattern results of related to the web-services using Parallel Evolutionary Algorithm (P.E.Algo) and our approach. This table shows the cumulative comparison of three approaches presented for Antipattern detections. Table4. 10 shows cumulative comparison of those web services that are used for antipatterns detection in three approaches. This table will helps us to generalize the results regarding success rate of Antipattern detection because data set is same for three tools.

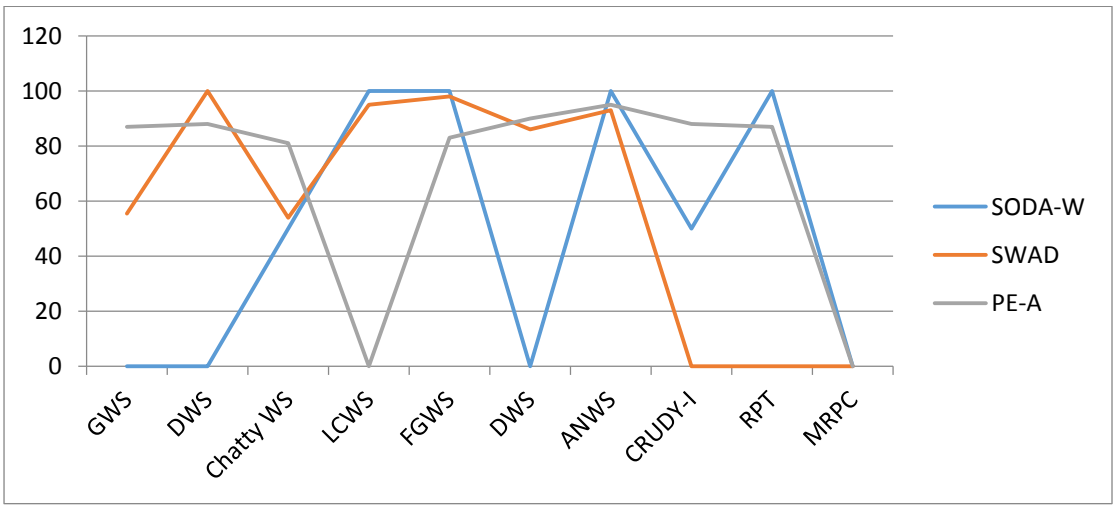


Figure 4.3Percentage of Antipatterns Detection for Three approaches

It can be seen from Table 4.10 that only one or two WS-related antipatterns are detected in each web-service. For instance, in the web-service named xOutlook only two antipatterns have been detected using P.E-Algo approach. Similarly, xMaster web-service has Fine Grained WS-related antipattern detected by P.E Algo technique.

On the other hand, SWAD is capable to detect more than two WS-related antipatterns if any present in the given web-service. Table 4.11 listed the names of the web-services in the first column and rest of the columns presents the number of antipatterns detected using SWAD approach.

Table 4.11 Comparison of Results Generated by SWAD

Services/Patterns	GOWS			DWS			CWS			MX			LCWS			RPT			ANWS		
	PE-A	SODA-	SWAD	PE-A	SODA-	SWAD	PE-A	SODA-	SWAD	PE-A	SODA-	SWAD	PE-A	SODA-	SWAD	PE-A	SODA-	SWAD	PE-A	SODA-	SWAD
AIP3_PV_Impact	X	X	√	X	X	X	X	X	X	X	X	X	X	X		√	√	X	X	√	√
Finding Service	X	X	X	X	X	X	X	X	X	X	X	X	X	X	√	X	X	√	X	X	√
XBATS	X	X	√	X	X	√	X	X	√	√	X	X	X	X	X	X	X	X	X	X	√
ExchangeRates	X	X	√	X	X	√	X	√	√	X	X	X	X	X	√	X	X	X	√	X	√
xAnalyst	√	X	X	X	X	X	√	X	√	X	X	X	X	X	X	X	X	X	X	X	√
X Master	X	X	√	√	X	√	√	X	√	X	X	X	X	X	√	X	X	X	X	X	√
Xoutlook	X	X	√	X	X	√	X	X	√	X	X	X	X	X	√	X	X	X	√	X	√
Xrelease	X	X	√	X	X	√	√	X	√	X	X	X	X	X	√	X	X	X	X	X	√
Xcompensation	√	X	√	X	X	√	√	X	√	X	X	X	X	X	√	X	X	X	X	X	√

GOWS: Gob Object Web Service , DWS: Data Web Service , CWS: Cruddy Web Service , MNR :May be its not RPC , LCWS: Low Cohesive Web service , RPT: Redundant Port Type, ANWS: Ambiguous Name Web Service.

4.5Conclusion

The detection of web service antipatterns from source code supports maintenance, refactoring and highlights poor practices adopted by developers during development of software applications. The detection of antipatterns from SOA is still young area. A limited number of approaches and tools are presented by different authors for the detection of antipatterns from SOA based software projects. The state of the art approaches are not flexible for code first and contract first concepts. Our proposed approach has three major contributions. First, we present customizable definitions and algorithms for detection of SOA antipatterns from multiple languages with varying features. Second, our approach is flexible due to application of SQL queries and regular expressions for matching definitions of antipatterns in the source code and these searching queries are not hard coded in the source code. Our approach is capable to detect eight SOA antipatterns from 7 weather related and 60 finance related web services. A prototyping tool is developed to validate the concept of approach. Thirdly, we evaluate our tool on two domains of web services implemented using different programming languages and recovered eight antipatterns with improved accuracy. The results of presented approach are compared with two state-of-the-art approaches. Our results illustrate the significance of customizable antipatterns definitions and lightweight searching techniques in order to overcome the accuracy and flexibility issues of previous approaches.

Chapter 5 Service Oriented Correction of Antipatterns for RESUTful APIs

5.1 Introduction

Service system evolution is quite different as compared to the software system evolution. The major complexity of service system evolution is due to the distributed nature of services. Components of services reside on different servers, organization and beyond the control of an individual entity. However, change in any service system components may not affect the direct change in its corresponding interface [154]. Lehman and Belady highlight the importance of evolution for software to meet end user requirement [155]. Moreover, the evolution of API can be assessed with the new version of software component [156].

In the perspective of statically linked APIs, Dig and Johnson addressed numerous changes in the interfaces with respect to time [156] and Laitinen addresses the high return on investment for the migration of new software version [157]. If we consider these issues in context of web API, developers cannot afford lethargy that was observed by Laitinen [157], a developer must follow the strict deadline from web API providers for migration towards a new version. It is also observed that for static environment developers may have an option to stay with the older version like libxml, which still meet the needs.

However, web service APIs are dependent on their providers that can change older version (and functionality) thus forcing API up gradation. On the other hand, another study revealed that only 4 APIs that were statically linked used out of the total of 1476 Source forge project [158]. This may imply that developers have no control over API evolution and client developers have to change their application code according to the new version [158]. Another survey among 130 Web API client developers reported a large number of complaints about current API providers like improper documentation, changes in API without warning message and preceding poor industry standards [159].

Web APIs are implemented by using two different protocols SOAP and REST. REST is well-known architecture style that is gaining popularity day by day. REST architecture style is mostly used amongst the top API providers due to the popularity of World Wide Web (WWW). REST uses HTTP and HTTPS together with URIs and MIME type with a simple and standardized availability for all kind of

platforms[160].Even more important is the effect of REST style on the quality attributes of software systems. Distributed software systems that follow REST architecture style implicitly support durability, inter dependency among the evolution of REST components, extensibility and scalability among others [6]. However, what will be the outcome when these REST API evolved? The main challenges in achieving these quality attributes are continuous evolution of REST API due to the change in user requirements.

All of the above mentioned studies addressed the changes in Web APIs due to user requirement and improper usage of industry standards. The rapid changes in user requirement also forced the developers to follow wrong practices called as antipatterns in contrast to design patterns that forced developers to follow industry standards [161][162][163].

There are some studies that reported the detection of REST design patterns as well as antipatterns. The detection of REST patterns and antipatterns with a tool support SODA-R first time introduced by Palma et al.[56]. SODA-R detected REST design patterns as well as antipatterns by using static and dynamic properties of REST APIs with an average recall about 94 percent from widely used REST APIs providers like YouTube, Facebook and Twitter [56]. Another study also detected the REST linguistic antipatterns with a tool support DOLLAR that examine the URI of widely used REST APIs [57]. A very recent approach detected the REST linguistic antipatterns by using semantic and syntactic [58] analyses. Another study examined three well-known REST APIs for their lexicon with a tool support, called CLOUDLEX that helps to extract and analyze REST cloud computing lexicon [164]. Moreover, they also analyze the designed aspect of famous REST API like Google cloud platform, OpenStack and Open Cloud Computing [120].All of the above-mentioned researchers focus only towards detection of REST design patterns and antipatterns.

While moving towards the solution of antipatterns for REST APIs, there is no study that reports an automated approach for the correction of REST APIs antipatterns. We are also not able to identify any study that reports the evolution of REST antipatterns from one version to another. Automated tracking of antipattern correction across each version also assists developers to track changes for the entire revision history of REST APIs. The result of this study will assists developers to assess the quality of their

client REST APIs that may undergo various changes in case there is a change by the major service providers.

However, there are many studies for object-oriented software engineering that addresses the evolution of code smells along evolution of software. It was also observed from a case study that quality of software deteriorates before and after transfer from one version to another [166]. The implication of code smells on code quality have been empirically assessed in recent study after examining version history of 200 open source projects [167]. This study investigated smells that introduce commit and finally, 10k classes were manually analyzed from those commits. Another study tracked the quality of mobile application along with their evolution with a tool support PAPRIKA [168]. PAPRIKA addressed the evolution of antipatterns along 106 versions of mobile application using their binaries [168]. However, there is no such study that addresses the evolution of antipatterns along the evolution of REST APIs after examining the versions of REST APIs. This study helps to report a clear guideline along with tool support for correction of antipatterns for REST APIs after examiningtheirversion or revision history.In this exploratory qualitative study, we try to answer the following research questions

RQ1: When Antipatterns are introduced in REST APIs?

We tried to investigate that in which circumstances these antipatterns are introduced in REST APIs.This may help to check that antipatterns are introduced as a result of maintenance in REST APIs.The answer to this research question also helps to report specific services that are constantly error-prone and have the most number of changes. Therefore, we investigated the presence of possible trends in the history of REST API versions for the evolution of a specific type of antipatterns.

RQ2: How Antipatterns are evolved?

In this research question, we aim to investigate how long antipatterns remain in the services. That is to say, we want to investigate survivability of antipatterns that is the probability that antipatterns instances survive over time. Evolution of each antipatterns along with versions or revision for each REST APIs are also checked.

RQ3: How do developers remove Antipatterns?

Evolution history will help us to monitor the changes for each antipattern across versions. There is a possibility that some antipatterns are removed or evolved again.

This question provides us a clear answer for the correction of antipattern approaches used by famous REST APIs providers after collecting real-time traces from year 2015 to 2017. This will also help to investigate that which refactoring operations are used by the REST API providers to remove these smells.

We selected 11 widely used REST APIs and collected their trace history using tool SODA-R over the past two years. Moreover, we track how they remove antipatterns and then extend SODA-R for the correction of REST API antipatterns. This tool is publically available and tested on 11 widely used REST APIs along with their available versions of 200 REST services and have precision 100% and recall 94 %.

5.2 Related work

A. Empirical Studies for RESTful web services

REST is gaining popularity in the web service community that introduces a new debate in industry related to the design of RESTful web services [169]. Li li et al.[169] noticed that most of the web services that claim to use REST architectural style are not hypermedia driven . This problem also makes REST not scalable, extensible and inter-operable as promised by its architectural style. REST chart model proposed by Li Ii et al [169] helped to design and describe the REST APIs without violating REST. This study helped to force the REST constraints automatically. Another study presented an approach to analyses the REST APIs based on machine-readable description [170].

REST API description language like Swagger and RAML also gaining popularity day by day and this was also confirmed from a recent study regarding Open API initiative to use these languages as API description. Machine readable API description can be applied at design time as it requires an API model, not its implementation [171]. This study reported the description of REST APIs into multiple languages to transform into canonical meta model. This canonical meta model acted as the repository to calculate metrics. This model is tested for 286 SWAGGER REST API. Results are presented based on the metrics proposed for the REST APIs evaluation [171]. Results showed that there was an average 9 to 40 resources per API but the distribution of resources among REST APIs were not even like Azure API uses 61.5% resources as read-only whereas the distribution of resources for Google was even. They also noticed that APIs are “more wide than deep” [171].

Another study showed that REST Resource model could be extracted from procedure oriented service interface [172]. WSDL documents contain various aspects of the service interface and the proposed approach mainly focused on the port type element of service interface for model extraction. The output of REST resource extraction model is the hierarchical model of resource type which contains primary informational entities that should be present in the REST API under analysis. However, this study did not focus all aspect of REST API models like media type, hyper media, and caching [172]. This model was further tested for Amazon simple storage service that store data both procedure oriented (WSDL) as well as resource oriented (REST) [172]. The technique was validated for the accuracy of intermediate steps for resource extraction, the quality of extracted resource model, productivity enhancement comparison with the manual process and time performance with respect to the service size. The result showed that intermediate models have accuracy between 88% to 96.6% that were calculated using 867 valid WSDL documents containing 12,918 operations [172].

A recent study reported the syntactic and semantic analyses of REST APIs for the detection of their linguistic patterns and antipatterns [58]. This study is validated with a tool called SARA (Semantic Analysis of RESTful APIs) that use WORDNET, Stanford CoreNLP along with Latent Dirichlet Allocation (LDA) topic modeling technique that helps to check the URI nodes to decide their semantic similarity [58].A new antipattern is reported by the name as Pertinent versus Non-pertinent Documentation. SARA is validated for 12 widely used REST APIs for the detection of their linguistic patterns as well as antipatterns with improved accuracy reported as 75% to 88% as compared to DOLAAR [58] that has average precision 79% for the detection of linguistic pattern and Antipatterns from REST APIS. Dollar tool analysed the syntactical URIs design problems as most of the REST APIs under analyses have unorganized URI nodes. The validation of DOLLAR tool suggested that most REST API providers use nouns in their resource name as compared to verbs [58]. The other study also reported the problem in cloud lexicon used by famous cloud providers like Google, Cloud Platform, Open Cloud Computing Interface, and Open Stack [164]. The verdict is validated with a tool approach called CLOUDLEX for extracting and analyzing cloud computing lexicon. It was observed that REST API under analyses are heterogeneous and there is no consensus among REST API providers regarding

term used for cloud computing lexicon [164]. There are different studies that empirically validate the REST APIs performance for mobile devices [173] and use of RESTful web resources practices for designing REST APIs [170]. However, above all mentioned approaches discusses either good or bad practices in REST API URI or detect antipatterns and design patterns but none of them discusses the evolution of REST API for the correction of antipatterns.

B. Studies on Antipatterns Correction

Many studies are reported that give an idea about code smell correction for Object oriented software engineering by using different techniques from the area of data mining, natural language processing, source code metrics and machine learning.

1) ***Antipatterns Correction for OOSE****:* There are a number of studies that discussed the correction of antipatterns by using various techniques to improve the source code quality. Simon et al. [174] reported that it is difficult to find the location of bad smells manually for correction. They argued that Metric-based techniques can be used for the selection of suitable type of refactoring. Their tool assisted developers to perform suitable refactoring operations by using metric based software visualization tool [174]. Ge et al. [175] invented a tool name Benefactor that suggests developers to perform auto refactoring and get rid of manual refactoring. Ivkovic and Kontogiannis [176] reported software architecture artifacts as the highest level of abstraction that also helps to map requirements to design. UML 2.0 is used as a model to perform architecture refactoring.Finally, software metrics are applied on UML model for the decision of most suitable refactoring operations.

Moha et al.[177] presented a methodology based on the meta model that helps to correct and detects high-level design defects using refactoring. Detection of design defects use source code metrics along with structural and semantic information of source code [177].Another study uses genetic programming for the detection and genetic algorithm for the correction of design defects [178].

While moving towards the recommendation studies that help to decide when to re-factor the code, Moha et al.[179][180] suggested approaches that use Formal Concept Analysis(FCA) and Relational Concept Analysis(RCA) to improve the code quality.Suitable refactoring operations are performed with the help of coupling and cohesion metrics values [179][180]. Design defects problems are resolved by

redistribution of the class members between new and existing classes. This may improve the cohesion and coupling problems. The authors [179][180] uses PADL (Pattern and Abstract-level Description Language) to model source code and Galicia v.2.1 to build and show the concept lattices. Another study reports the use of RTM (Relational Topic Model) to analyze the structural and textual information from software that support move method refactoring [181]. This approach is also validated by two case studies and results show that RTM recommendation is quite useful that also helps to improve the code quality and remove feature envy smell [181].

A recent study proposes a technique with a tool support that builds an agent to autonomously perform refactoring for a code project [182]. This technique helps to identify code smells, need of refactoring and the behavior of the code after refactoring. They apply extract class, move method and encapsulation field refactoring techniques to remove feature envy, public field and data class code smell. Test cases are written and new test cases are generated before and after the refactoring. Saferefactor an eclipse plug-in is used to generate new test cases for the evaluation of auto refactoring [182].The authors also identified multiple problems associated with previous approaches like a manual implementation of rule card. The above-mentioned approaches just consider the improvement in the quality of the software by reducing a number of existing smells.

Another research reported Non-dominated Sorting Genetic Algorithm (NSGA-II) for the detection and correction of software defects that claim to minimize the time used for refactoring [90]. This technique automates the detection rules by using genetic programming to get rid of manual rule implementation having an average precision of 85 percent. Correction of code smells using development history also used NSGA-II [12]. Moreover, NSGA-II also minimizes the time needed to correct a defect with minimal efforts [79]. It is also observed that refactoring could not be applied at all stages of software development that may have the negative impact on source code quality. This might be due to the inexperience developers that use refactoring to remove smells without considering its importance [86]. A recent study reported a framework called monitor based instant software refactoring that helps inexperience developers to perform more refactoring with the help of monitor running in the background. This monitor warns the developers only for those changes that may introduce code smells. Feedback from developers are also used to optimize smell

detection algorithms [86]. This framework was also tested and provides 140 % increase in code smell detection and thus reducing code smells by 51 % [86]. Another study also discussed the use of correct refactoring options that may help to improve the system performance as compared to those options that might have the negative impact on system evolution [183]. The approach is automated with JDeodrant an eclipse plugin that ranked suitable refactoring options to remove smells. The proposed approach is also evaluated on JMol and JFreeChart. Few researchers used implicit dependencies as a technique for refactoring [184].This technique guided the developers to use most suitable refactoring operations on a given context using Atrributed Graph Grammer (AGG) transformation with a combination of critical pair algorithm running in background [184] .

2) ***Correction of Antipatterns for Web services:***Pautasso [185] reported the solution of different REST antipatterns inform of guidelines and there are also different books available that focus towards the improved service discovery by avoiding Antipatterns in REST APIs [186][187][188][189]. However, no such technique is available as per best of our knowledge that focuses towards the correction of REST antipatterns. Moreover, developers using web APIs are forced to meet the rapid changing requirement from a client as well as APIs providers. A recent study reported the APIs evolution and the effects of this on source code quality [190] through a qualitative study of the Facebook, Google Maps, and Twitter. They conducted a case study to know the information regarding the evolution of APIs. However, there is no discussion about the solution of the problems that these APIs providers face due to the evolution [190]. Daigneau in his book [191] addressed the importance of design decisions for using web services as technology is changing with respect to time and also reported changes that are important in different application scenario. However, focus was only for the suitable pattern choices [191]. Furthermore, most of the web API providers allow their developers to use old versions for extended time period. Authors also focus to use the blackout tests for change analysis and report fewer changes in APIs with respect to time [191].

While moving towards the detection of static code problems in web services, there are different studies that reported the violation of different design principles in web services commonly called as Antipatterns [78] [48]. Software industry mostly uses the

code-first technique that first generate source code and then use WSDL generation tool.

Matoes et al. [78] proposed a tool that assists developers to generate WSDL document with few Antipatterns. The author uses text mining with the help of meta-programming and provide an Eclipse plugin called AF-Java2WSDL(Antifree-Java 2 WSDL) [78].The author also reported an approach used by the contract-first WSDL document [48].Another study reported that traditional object-oriented metrics-driven refactoring for the code first WSDL service. Authors reported different threshold of various source code metrics that effect the number of occurrences of WSDL Antipatterns [49].It is also observed that early code first refactoring technique helps to avoid antipatterns during migration phase from traditional systems to SOA [192]. This approach also helps a community to reduce the migration cost from traditional systems towards SOA [192].The strategies to detect the antipatterns from WSDL is either contract first or code first. The correction of WSDL antipatterns are based on OOSE refactoring techniques [192]. Another research also proposed the use of early code refactoring for the code first approach that helps to retrieve the syntactic registries [100].

Use of CBO, WMC, ATC, EPM metrics for service refactoring is studied by using four different java to WSDL tool and two different registries [100]. The experimental results proved that refactoring options decrease the WSDL antipatterns independently of the WSDL generation tools for effective service retrieveability [100].There are number of approaches that highlight the importance of improvement in WSDL document by highlighting the bad practices that prevent the discovery of any web services [193]. Removal of the antipatterns in WSDL may improve the performance. This study is also evaluated by 26 professionals regarding improvement in service discovery. However, no automated process is adopted for the correction of WSDL antipatterns. Another study also reported the WSDL bad practices commonly found in WSDL document and suggested some solutions for these Antipatterns [194]. There is also an attempt towards the discovery of static and dynamic analysis of antipatterns for SOAP services by Francis *et al.*[56] with the help of tool support SODA-W. Palma*et al.* [56] used DSL and Frascati runtime support for static and dynamic analysis having an average precision of 89%. Another study also addressed the

antipatterns detection for SOAP services using PE-A (Parallel evolutionary algorithm) derived from genetic algorithm [54].

5.3 Study Design

This empirical study aims at investigating the evolution of REST antipatterns and proposes an automated approach for the correction of antipatterns in RESTful APIs. More specifically, this study addresses the following three research questions:

- RQ1: When antipatterns are introduced in RESTful APIs?
- RQ2: How antipatterns are evolved for a specific RESTful API?
- RQ3: How REST antipatterns are removed from RESTful APIs?

Palma *et al.* [56] developed SODA-R tool for the detection of REST antipatterns in RESTful APIs. The authors showed high detection accuracy (*i.e.*, with an average precision of 89.42% and an average recall of 94%) on popular APIs like YouTube, Twitter, and Facebook. To answer the three research questions stated above, we rely on the SODA-R tool [56]. The motivations for using SODA-R are as follows:

- In the literature, SODA-R is the only tool that can automatically detect the REST antipatterns by dynamically invoking a REST service;
- SODA-R supports invoking a specific version of a RESTful API relying on FraSCAti service component architecture [55] for the detection of REST patterns andantipatterns;
- SODA-R generates traces that can be reused by the engineers for detailed analysis of HTTP requests and responses of RESTful APIs.

The focus of thisstudy is to examine the evolution history of seven REST antipatterns from the literature [196]. The short description of these antipatterns are as follows:

1) **Breaking self-descriptiveness** occurs when developers ignore the standardized headers, formats, or protocols and use their own customised ones, which breaks the self-descriptiveness or containment of a message header. This poor practice also limits the re-usability and adaptability of REST resources [196].
2) **Forgetting hypermedia** refers to the lack of hypermedia, *i.e.*, not linking resources, which hinders the state transition for REST applications. One

indication of this antipattern is missing URL links in the HTTP response or resource representation that restricts clients to follow the links [196].

3) **Ignoring caching** occurs when REST clients and server-side developers avoid the caching capability. The caching capability is among the principle REST constraints. The developers may avoid caching by simply setting the value for Cache-Control to ‘no-cache’ or ‘no-store’ without an ETag in the response header [196].

4) **Ignoring MIME types** refers to the practice that the server side developers often have a single representation for REST resources or even rely on their own customized formats that may limit the resource (or service) accessibility and re-usability. In general, the requested resources should be available in various formats, *e.g.*, XML, JSON, HTML, or PDF [196].

5) **Misusing cookies** occurs when client developers send keys or tokens in the Set-Cookie or Cookie header field to the server-side sessions that concerns both security and privacy. In REST, session state in the server side is not allowed, and use of cookies strictly violates the principle of RESTfulness [196].

6) **Ignoring status code** refers to the practice when REST developers avoid the rich set of pre-defined application level status codes suitable for various contexts and rely only on common ones, namely 200, 404, and 500, or even use the wrong or no status codes. However, using the correct status codes from the classes 2xx, 3xx, 4 xx, and 5xx may make the communication between clients and servers semantically richer [196].

7) **Tunneling through GET** occurs when the developers rely only on GET method for all sorts of actions including creating, deleting, or updating resources. However, the HTTP GET method is inappropriate for every actions other than retrieving resources [196].

8) **Tunneling through POST,** similar to Tunneling through GET, occurs when the developers depend only on HTTP POST method for sending all sorts of requests to the server including retrieving, updating, or deleting a resource. In REST, the proper use of POST method should be limited in creating server-side resources only [196].

The following sections discuss our data gathering process and how we analyse to answer each research question.

5.3.1 Data Extraction and Analysis

To answer RQ1 on *when antipatterns are introduced in RESTful APIs?* We performed the detection of REST antipatterns using SODA-R tool [56] on the RESTful APIs reported in Table 5.1. The detection results reported in [56] are from 2015. Thereafter, we rerun the detection and check the occurrences of antipatterns again for the year 2017 by using SODA-R for the same set of RESTful APIs. This also helped us to study and compare the occurrences of REST antipatterns for a specific RESTful API. We also studied the change history for 12 RESTful APIs from their online changelog as listed in Table5.1.

As we observed from the study by Palma *et al.* [68], services that are involved in antipatterns usually are more change prone than services that are part of design patterns. Therefore, software components involved in antipatterns face higher structural changes as compared to the software components involved in design patterns [197]. We also observed the occurrences of antipatterns that are newly introduced or disappeared for a specific period of time. However, the techniques used to measure changes in Object-oriented (OO) systems like *code churn* and *changeproneness* by different studies [113] [198] cannot be applied directly to RESTful APIs due to several reasons:

- Most of the RESTful APIs do not provide version history. For example, YouTube has its Version 3 running since 2013. According to the software systems versioning guideline, any change in software systems must be reflected in its system version number [199], [200], [201]. Software versioning also assists in backtracking bugs and issues in software systems to their particular release [202]. In our study, in the absence of versioning support by RESTful APIs, we collected their online changelogs. For example, the APIs YouTube and Alchemy do not carry their version information in their HTTP request URIs but in their online changelog only. On the contrary, StackExchange explicitly shows it API version in its request URIs.
- One possibility to identify the changes by using commit history [167]. In reality, commit history are also not available from RESTful APIs providers.

Moreover, RESTful APIs providers not always provided changes for each version online. Therefore, for such services we can only rely on their online documentations. We manually validated for the changes in the instances of REST antipatterns based on their changes in online documentation. We also checked the migration date for RESTful APIs and further validated changes from their trace history over the two years using the tool SODA-R [56].

- We also observed some RESTful APIs where we can find different API versions running at a given time, for example, StackExchange and Facebook. These APIs accept HTTP requests to its multiple versions. This can also help us in preparing a concrete definition for the correction of REST antipatterns.

Therefore, RQ1 is answered based on the changes in the request URIs and HTTP requests and responses associated with the detection of REST antipatterns for an API for the year 2015 and 2017. If we observe changes in the total instances of antipatterns or instances of antipatterns for a specific RESTful API then we check those differences with their associated traces that may provide the hint or reason for the appearance or disappearance of antipatterns. The detection of antipatterns from version history for the years 2015 and 2017 provided a timeline regarding effects of antipatterns for specific service. Figure 5.1 shows the steps related to each researchquestion that cover the correction process starting from the detection of antipatterns, collection of trace history, and the evolution of antipatterns in RESTful APIs.

Table 5.1 List of RESTful APIs under Analysis

RESTful APIs	Available Version	Monthly Active Users	Online Changelog URLs
Alchemy	Version 1	*not available*	https://www.ibm.com/watson/developercloud/doc/index.html
Bitly	Version 3	13,530	https://dev.bitly.com/api.html
Charlieharvey	Version 1	*not available*	https://charlieharvey.org.uk/about/api
Dropbox	Versions 1 and 2	500 million	https://dropbox.github.io/dropbox-api-v2-explorer
Musicgraph	Version 2	1 billion	https://developer.musicgraph.com
Ohloh	Version 1.0	669,601	https://www.openhub.net
Facebook	Version 2.3 to 2.10	1.94 billion	https://developers.facebook.com/docs/apps/changelog
Instagram	Version 2	319 million	https://instagram.com/developer/changelog
Twitter	Version 1	600 million	https://dev.twitter.com/ads/version/1/changes-in-version-1
Teamviewer	All versions	300 million	https://oldteamviewer.com/download/changelog/windows
StackaExchange	Version 2.0, 2.1, 2.2	345 million	https://api.stackexchange.com/docs/change-log
Youtube	Revision history from 2013 to 2017	1 billion	https://developers.google.com/youtube.v3/revision history

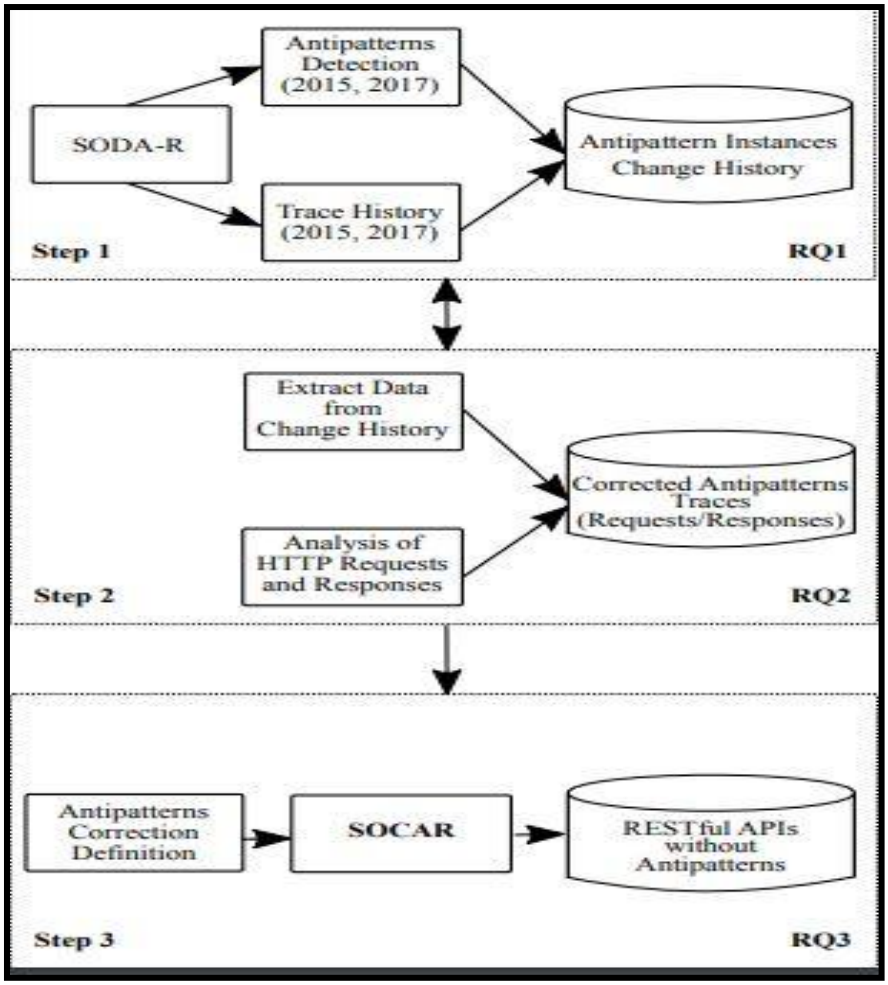


Figure 5.1 Research Methodology

To answer RQ2 on *How antipatterns are evolved for a specific RESTful API?*

We need to know when antipatterns are detected by SODA-R [56] for each RESTful API. We collected the results of recent changes in each RESTful API for three consecutive months to identify the changes in theirtraces and instances of antipatterns. This process helped us to mine the changes in RESTful APIs and formulate the definition of correction. Moreover, since SODA-R detects antipatterns instances dynamically and there could be changes in RESTful APIs requests and responses each time SODAR calls, we collected the trace history and prepared a list of request and response parameters returned by RESTful APIs to formulate a correction definition used by the RESTful APIs. If there is an instant change in request/response of a RESTful API then there is a need to check the changes in the API's changelog to identify which parameters or methods the developers introduced that may cause the (dis)appearance of antipatterns. However, this step is performed only for those RESTful APIs that change frequently. A high number of changes associated to a

RESTful API suggest the motive on service evolution [154]. Finally, being inspired from the study of API evolution from [203], we tried to identify the effect of antipatterns on a RESTful API's evolution.

Table 5.2 Operations Performed in Evolution

Operation Name	Actors	Descriptions
Fixing Error Code	Response	API's response status change.
Enhancement	Request, Response Body	API changes its version or domain.
New Feature	Request , Response	API has an additional functionality.
Refactoring	Response	Correction of Response through the addition or deletion of attributes

After completing the process, we prepare a list of factors as suggested in [167] and assign tags to each change in the traces and changelog for a RESTful API. Table 5.2 highlights the tags and their roles in API evolution.

To answer RQ3 on *How REST antipatterns are removed from RESTful APIs?*

We rely on the output of RQ2 in Step 2. Results obtained from RQ2 indicate changes associated with each RESTful API's HTTP requests and responses. Later, these changes can be verified using their traces that provide more insights on different attributes for the correction of REST antipatterns used by the RESTful APIs providers. In an empirical study in [66], authors observed relationship between code smells and design patterns. Thus, removal of specific antipatterns by the RESTful API providers may stimulate the use of design patterns for some REST APIs. We also observed the total number of instances of antipatterns and design patterns found for this study and make a list of corrected requests and responses used by the service providers to investigate the evolution of antipatterns in each RESTful API.

We mined all the changes in HTTP request/response headers and bodies for each RESTful API for two years starting from 2015 (*i.e.*, when the SODA-R [56] tool is developed) and then restudy the traces in 2016 and 2017. We also observed the variations in HTTP requests/responses for three consecutive months (in 2017) to observe any changes in results of antipatterns evolution through their detection traces. The benefit of mining trace history includes the proof of the appearance and removal

of antipatterns as well as changes implemented by RESTful API providers. We consider:

- 11 RESTful APIs correction results along with their traces;
- Analysis of most suitable operations performed for the correction. The use of corrected request/response and body parameters will help to implement the most suitable correction operation as reported in Table 5.2;
- According to the definitions of REST antipatterns [56], clients might requests resources in various representations from the server, and the REST service providers must fulfil any client requirements. Thus, if there is any such change, *e.g.*, change in resource representation format, it can come in the form of enhancement, and as a new feature if the core functionality is changed;
- Online trace history helps to prepare the list of refactoring operations used by the RESTful API providers after considering their changelog, documentation, and antipattern evolution results.

5.3.2 Correction of Antipatterns using Evolution History

This section explains the steps we performed to acquire the evolution of antipatterns for RESTful APIs.

An Example of Ignoring MIME Types Antipattern: We invoke seven HTTP methods from Alchemy API and collect their requests and responses from 2015 to 2017. We discuss here one HTTP call using the /calls/url/URLGetText and for the changes in the evolution history for *Ignoring MIME Types* antipattern [56]. This anitpattern is introduced when the REST developers do not consider multiple resource representations for a resource which may force client developers to manage their resources in a single format, for example, in JSON or XML. Figure 5.2 shows the change history of traces collected using the SODA-R tool [56] for the detection of *Ignoring MIME Types* antipattern in Alchemy API for the callusing /calls/url/URLGetTextSentiments. Figure 5.3 shows the real time traces of Alchemy API that reports the removal of ignoring mime type antipatterns due to the migration on Watsonplatform. Figure 5.4 reports the correction algorithm implemented in proposed approach. The complete tracelog is available on our project website[33].We

[33]www.sofa.uqam.ca

collect and verify such heuristics from trace logs of antipatterns and then implement them in SOCAR (Service Oriented Correction of Antipatterns in REST) for the automatic correction of REST antipatterns like *Ignoring MIME Types*. We consider Alchemy API as our example because this API is migrating their services from HTTP to HTTPS by changing its URIs from www.access.alchemyapi.com to www.gateway-a.watsonplatform.net. Thus, we expect some improvements and changes in the API, and, as expected, the *Ignoring MIME Types* was present and then disappeared after the migration phase. The results of this tracelog can also be observed in our trace history from July 2017 which shows the information in the response body of an HTTP call. A research study conducted for Web services migration also discusses the challenges that REST client developers face during the migration phase by service providers [190]. However, it is worth mentioning that each RESTful API might have unique migration plan across versions with varying effect on their clients. Changes during the migration may include changes in calling methods, protocols, or changes we notice here in Alchemy. We show the heuristic in Figure 5.4 that we applied for the correction of *Ignoring MIME Types* REST antipatterns.

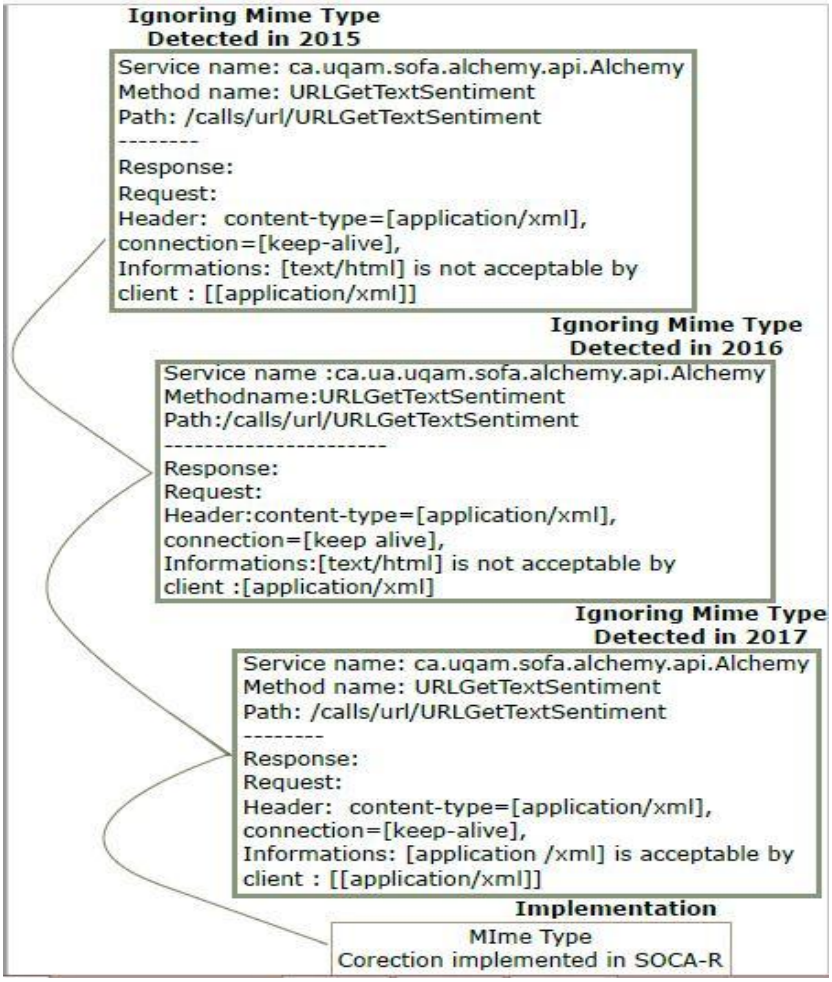


Figure 5.2 Migration Traces of Alchemy RESTful API

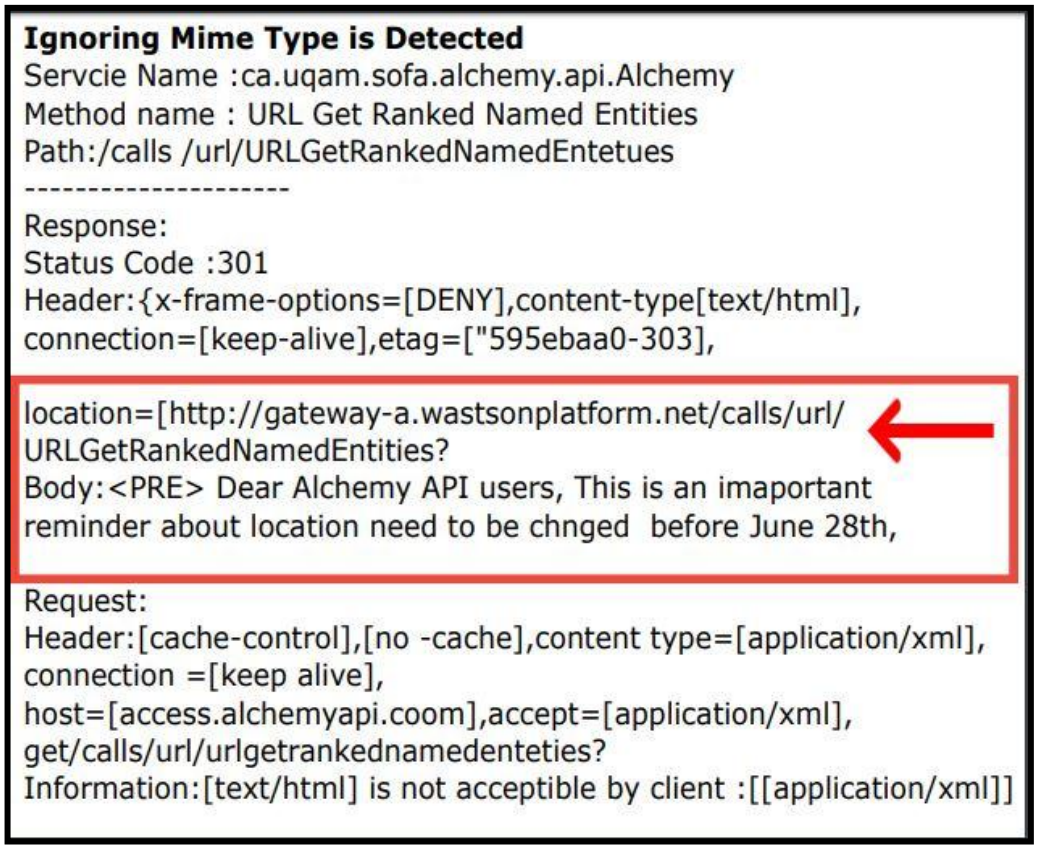


Figure 5.3 Migration Traces of Alchemy API

Algorithm 1: Correction of *Ignoring MIME Types* REST Antipattern.

Input: *request-metadata, response-metadata*
Output: *Content Type Design Pattern*
request-metadata ⟵ *xml, json, pdf, html, ...;*
response-metadata ⟵ *xml, json, pdf, html, ...;*
if *request-metadata.containskey()= "accept"* **then**
  if *request-metadata* ≠ *response-metadata* **then**
    responsemetadata.add (requestmetadata);
    "Ignoring MIME Types Corrected based on Client Request"
  end
  "Ignoring MIME Types Instances Removed"
end
"Content Type Pattern Detected"

Figure 5.4 Correction Heuristic of Mime Type Antipatterns

Second example of Forgetting Hypermedia Antipattern:

Algorithm 2: Correction of *Forgetting Hypermedia* REST Antipattern.

```
Input: http method, entitylinks format, entity location
Output: Entity Link Design Pattern.
http method ←— GET, PUT, POST, DELETE;
entitylinks format ←— xml, json, pdf, rdf, html, ...;
entity link←—link,location;
if response.getstatus()= "4K" Or response.getstatus()=
 "5K" then
 |  remove from detection and correction ;
end
if response.getBody() and response.getmetadata() ≠ Null
 then
 |  if checkLinksBody()←—xml,json,pdf,rdf then
 |  |  if checkLinkMetaData()←—location,link then
 |  |  |  Link detected ;
 |  |  |  if
 |  |  |   checkLinkMetaData().contains≠location,link
 |  |  |   then
 |  |  |   |  metadata.add(Add URL dynamically);
 |  |  |  end
 |  |  end
 |  end
end
 " Forgetting Hypermedia Antipattern Correction
Adding link in response meta data"
"Entity Link Pattern detected"
```

Figure 5.5 Correction Heuristic of Forgetting Hypermedia Antipattern

The *Forgetting Hypermedia* occurs with the absence of URL links in the HTTP response that restricts the usability feature of RESTful APIs. Hypermedia is the concept of linking resources, *i.e.*, a set of connected resources where applications move from one state to another [56]. The API developers design and rely on resource URIs but the client might not be able to receive and follow such links because the server never expose them via responses. Ideally, REST developers must provide at least one URI link to avoid this antipattern. Client developers can ask resources or links in various forms and server should provide as per client request. The responses provided by the server may combine certain attributes like meta-data or

location of links, and while processing the response client developers check each link and its resource types. The evolution of this antipatterns can be considered as the changes in responses in terms of changes in *location* attribute or formats in the responses provided by the REST API providers. We found slow change for YouTube as compared to Facebook and Dropbox. Moreover, Forgetting hypermedia instances are not evolved in Bitly and Alchemy API. Forgetting Hypermedia antipattern is found in YouTubein /guidecategories,/vidoes,/playlists for 2015 but evolved from /playlist service to the /subscription service in 2017. The total number of instances remains same for Forgetting Hypermedia antipattern. Therefore, total number of instances were 3 for year 2015 to 2017 but they were evolving from one service to another. Figure 5.5 shows the algorithm used for the correction of Forgetting Hypermedia antipattern.

Addition of location and link in metadata is done by considering the format of each request and fulfill the client by providing response that append the link and location attribute to the generic ”google.com”. This will also help client API or client to search specific terms in case there is no link provided by the service providers.

5.4 SOCA-R(SERVICE ORIENTED CORRECTION OF ANTIPATTERNS for REST APIs)

We used the SODA-R to cover the entire evolution history of antipatterns detection for REST web services. The general structure of SOCA-R is based on SODA-R approach. Figure5.5 describes correction heuristics used for the correction of forgetting hypermedia antipattern . The correction heuristics are implemented in SOCA-R, a new handler added in SOFA framework for the correction of REST antipatterns. SOCA-R uses Frascati Service Component Architecture for the dynamic invocation of REST API and biding each REST API client to the *FrascatiIntentHandler* for authentication of client request and response of each REST APIs. The complete methodology used for correction of RESTantipatterns is described as below

Step 1 :**Input:** Dynamic invocation of client request using service interface

Output : Instances of design pattern and antipattern

Description : SODA-R proposed by Palma *et al.*[56] used for the detection of antipatterns from REST APIs. SODA- R has implemented the detection heuristic of

different REST antipatterns for popular REST APIs like Facebook,YouTube, Alchemy and Dropbox. There is a need to check the antipatterns instances before implementation of correction heuristic.This step will help us to identify the increase or decrease in REST API antipatterns. This will also help to check the correction of specific antipatterns by REST API providers. We collected the total number of instances for specific antipatterns for 11 REST APIs. We performed the detection using SODA-R for two months continuously to identify the change in the detection of REST antipatterns.

Correction always needs a proof that services are improved and number of antipatterns are either decreased or removed. A study has been conducted that shows the relationship between code smells and design patterns [66]. Therefore, if the services are improved then total number of design patterns for a specific web service may also improved after the correction of antipatterns. We performed detection of REST patterns to check the possible number of design patterns used by each REST API providers. This step was also performed iteratively for two months and detection of possible design patterns instances were recorded that helped us to verify the correction of antipatterns and their possible effects on specific instances of REST design patterns .

Step2: Input

Refactoring operations and attributes values from trace history

Output

Possible refactoring consideration.

Description: Trace history of the design patterns and antipatterns are used to decide the possible refactoring operations used by REST API. Table 5.2 reports the major refactoring operations used by the service providers for the correction of REST APIs. As per the guideline of IETF [205], SODA- R implemented the list of standard request, response and status codes. The properties used to correct REST APIs antipatterns also available in the literature [56] . However, we also identified the list of non-standardized header description [205] and use remove refactoring operations if non-standardized header was detected. The non-standardized header was finally replaced with correct-standardized header. We also checked the complete trace history of each REST API to identify the mostly used non-standardized header. The

detail results of this step are reported in the result section. For the correction of Status Code problem, we implemented the status code list in the correction definition of Ignoring Status Code antipatterns and used add refactoring operations in response header field that shows the client correct status of REST APIs.There was a possibility regarding wrong status description so this error is corrected after applying replace method and removing wrong error description with the correct description available in our tool .All possible refactoring operations are implemented in SOCA-R for the correction of REST antipatterns.

Step3:

Input:

SOCA-R algorithm implementation

Output:

Design pattern Instances and antipatterns removal.

Description: REST APIs are changing day by day and this can also be assessed from their online change history available in Table5.1. These changes may affect their usage of multiple methods available for specific service. We implemented the correction heuristic based on the available definition in the literature [56] [68]. However, we also took the benefit of SODA-R real time traces and improved the definition of correction heuristic reported in the literature [56]. For the application, we followed the SOFA framework that enable the detection of antipattern dynamically. We automatically applied the correction algorithms as reported in Figure 5.3 and Figure 5.4. We added a new *REST Correction* handler to the underlying framework of SOFA. The list of correction heuristic used for each antipatterns is reported in Table 5.3. It is worth mentioning that quality of the system under analysis must be improved before and after refactoring. We implemented the correction heuristic in a way that improved the REST APIs quality of services and provide response as per client need. These heuristic are added by (1) Wrapping each REST API with an SCA component for the detection and then dynamically (2) adding missing attributes like content-type, response format, links, status code description to full fill the client needs and (3) automatically correction of antipatterns by removing their instances.

It is worth mentioning that after removal of Ignoring Mime Type, Ignoring Cache and Forgetting Hypermedia antipatterns the instances of Content Negotiation, Response

Caching and Entity Linking design patterns detected respectively. The results of correction heuristics were saved in csv file for further calculation of precision and recall.

The results were verified from traces collected dynamically that helped to identify the reason of antipattern removal from REST APIs. These traces also helped to identify the possible reason of removal of antipatterns for specific APIs like Alchemy has removal of Mime-Type antipatterns due to the migration of *www.gateway-a.watsonplatform.net* and Facebook also not report Mime Type antipattern due to the migration from *version 2.3 to 2.4.*

Table 5.3 Refactoring Operations Performed for each Antipattern

Sr.No	Antipattern Name	Properties	Refactoring Operations	Effect
1	Ignoring Mime Type	Accept, Content Type	Add Content-Type	Content Negotiation Pattern
2	Ignoring Cache	Cache-Control, ETag	Add Cache Control, Generate Unique E-Tag for Request	Response Caching Pattern
3	Forgetting Hypermedia	http-methods, entity link, location	add links, metadata info, status code	Entity Link Pattern
4	Breaking Self-descriptiveness	Request-header field, response-header field	Remove non standardized Header from Response	Antipattern Remove
5	Ignoring Status Code	http method, status, standardized status code description,	Status code number and description change, replace method for code description and number	Antipattern Remove
6	Misusing Cookie	Cookie ,Set Cookie	Remove set-Cookie, cookie from response metadata	Antipattern Remove
7	Tunneling Through post	http-method,request-uri	remove access,update and delete from resource uri	Antipattern remove
8	Tunneling Through get	http-method,request-uri	remove access, update and delete from resource URI	Antipattern remove

Similar information was also found for Dropbbox that has deadline of shifting fromversion 1 to version 2. However, StackExchange is following different strategy as they haveall 3 version running simultaneously i.e., 2.0,2.1,2.2. We have collected the trace history of all three available versions to define the correction heuristics. These heuristics can also be used for antipatterns correction from others APIS. These traces helped us for the verification of correction results for REST antipatterns. Figure 5.3 shows the migration of Alchemy API to the new location. We also found similar traces for Facebook migration from one version to another version. However, no such information is found for the Dropbox traces. Moreover, Dropbox APIs shows migration information at their developer's login account page.We were also not able to found such information for YouTube having version 3 running since last 4 years and also have quite long change history as reported in Table5.1.

5.5 Analysis of Results

5.5.1 RQ1: When antipatterns are introduced?

Antipatterns are introduced due to the continuous maintenance activities when developers try to provide on-time product. The evolution of antipatterns could also be seen in the Table 5.4. This is worth mentioning that total number of antipatterns in all REST APIs were increased in 2017 as compared to 2015. The antipatterns reported for Facebook are increased as compared to others APIs. Amongst the selected REST APIs testing datasets, Facebook has a long version history from year 2015 upto now. The resultsof antipatterns detection for Fcebook in 2015 are from version 2.3 and in 2017 version 2.10 is also available for users. We also observed no change in"Misusing Cookies antipatterns" because we are unable to find any instance of this antipatterns in 2015 as well as in 2017 for Facebook APIs.

We found constant variation in the result of Breaking self-descriptive antipatterns that constantly increased or decreased. Music graph, Facebook and Alchemy are those APIs that report the high increase in these antipattern in 2017 as compared to zero in 2015 for Breaking self-descriptive antipattern.

We were also unable to find variation in ignoring status code antipattern for the entire revision from 2015 to 2017 as compared to slow variation in tunneling through get antipattern. We have found very little change in Charlihavery API that only shows changes in breaking self-descriptiveness and mime type antipatterns. As the representation of content-type antipatterns are not changed that stop the growth of mime type antipatterns. Most of the APIs are still running with single mime type and content representation. We also checked the results of antipatterns detection of 11 REST APIs after calling specific version of service interface.

Stack exchange results showed changes in Breaking Self descriptive antipattern for all three versions available. Table 5.5 shows the antipattern detection result for 3 versions of Stack Exchange and two different versions of Facebook API. While moving towards the identification of most effected REST API, we checked the evolution of antipatterns of all the web services reported in 2017 from widely used REST APIs. As different service providers not completely maintained their changelog history so for the paper we are highlighting the evolution of antipatterns in YouTube,Facebook,Alchemy and Stack exchange due to the following reasons,

Stack Exchange has a complete changelog for all three versions also available on line and user can use any one of them. Facebook has 7 versions running simultaneously with a two-year time period from REST API providers for changing one version to another. Currently they are using version 2.10 but still version 2.4 is running and giving time to their client REST APIs for changing to the new version. Alchemy has version 0.9 running and the previous version is not available once they change to the new version. Regarding, version history we just checked the detection for version 0.9 and compared these results with the results of [56] for REST API antipatterns detection. We checked the relative change in the antipatterns for year 2015 to 2017 by using the formula described as under:

$$RelativeChange(x, x_reference) = \frac{ActualChange}{x_reference}$$

The results of Table5. 4 show relative changes in instances of antipatterns in year 2017 as compared to year 2015. The maximum change was seen in Facebook, Twitter and Bitly as compared to YouTube that has only 76% change. Antipatterns detection result of version history of StackExchange showed us the change of antipatterns instances from one version to another. Forgetting Hypermedia antipatterns are

constantly increasing from one version to another. The correction heuristic of Forgetting hypermedia antipatterns is also reported in Figure 5.5. The major cause of this antipattern is the violation of service response. Each service response must haveany link or use of link attribute in location. Most of the service API providers used single response format like json,xml or rdf but client API or user can ask response in any format as per the principal of REST [6].Correction of this antipattern must consider the response returned by the service API providers and match it with client API request or user request. Then we generically built parser that checked the child and parent node in the location or link attribute and if there is no link presented in both nodes then application provide link to the client generically. This is maintained by using import org.codehaus.jettison.*; that use JSON Array, Json Token and xml.parsers. Document Builder (DB), org.xml.sax.InputSource and Document Object Model (DOM)from W3C to parse each request as per service provider format and then complete the client request as per its demand. Location or Link is dynamically added after checking absence of link in location attribute.

While checking the REST APIs that are evolved for StackExcahnge version 2.0 to version 2.2. We found significant changes in the Forgetting Hypermedia antipattern like ”2.0/me/write-permissions” has no instance of this antipattern but for the next version this API reported one instance of Forgetting Hypermedia. The instance of this antipattern was not removed also in version 2.2 of stack exchange. We also noticed that there is an increase in the Forgetting hypermedia from version 2.0 to 2.2. This information could also be checked from the trace history of Stack Exchange API. We also found one REST service from StackExchange”2.0/question/featured that has no instances of Forgetting Hypermedia antipattern for version 2.0 to 2.2. StackExchange also provides versioned call for each of its service that helps users as well as client APIs to check the change in its services. However, Facebook has long version history but provides unversioned call and they gave suitable time to their client REST API providers from shifting old version to the new version. This information is also available in their online changelog.

A recent study also proved this information regarding Facebook unversioned call [190]. The changes in their REST services are referred to as migration not as changes into the version [190]. Palma *et al.*[68] invoked 65 services for Facebook graph API in 2015 but some of these services were not available due to extensive migration.

Therefore, we only considered those services that are still available for version 2.2 to version 2.10. These options helped us to identify the relative changes in antipattern evolution history. Un-version call of Facebook was also validated by the trace history collected after calling each service.

The trace history of the REST services is also available on our SOFA website for year 2015. As per the previous study [190] on web services, we don't use same procedure of checking the antipattern instances for each version of Facebook but we checked the online changelog of Facebook reported in Table 5.1. The Facebook services are constantly evolving and we found some services out of 67 from year 2015 that are not available and deprecated with respect to the time. We randomly selected 21 services and collected their trace information. We checked the presence of antipattern for these services.

We selected only those services that are available in version 2.10 just to make sure that they are constantly evolved from version 2.2 and were available for year 2015 and also for version 2.10 for the year 2017. It has not been possible to check each version due to the dynamic nature of request and response because Facebook support unversioned call. Traces of version number is not found either in CSV file or in trace log.

Table 5.4 Evolution History of REST APIs

Sr.No	API Name	Year	BSD	FHM	IMT	ISC	IC	MC	Tun	RC (%age)
1	Alchemy	2015	0	1	2	1	7	0	5	94
		2017	9	1	2	1	9	0	9	
2	Music Graph	2015	0	1	2	1	7	0	5	94
		2017	9	1	2	1	9	0	9	
3	Bitly	2015	0	2	3	0	0	0	2	243
		2017	0	5	15	0	0	0	4	
4	DropBox	2015	12	9	0	0	12	0	5	18
		2017	17	14	0	0	8	0	6	
5	Twitter	2015	10	3	9	0	0	0	0	225
		2017	25	6	25	6	14	0	2	
6	YouTube	2015	9	3	9	0	0	0	0	76
		2017	17	3	14	0	3	0	0	
7	CharliHavery	2015	4	0	4	0	0	0	0	50
		2017	12	0	0	0	0	0	0	
8	Zappos	2015	7	0	0	0	0	0	0	67
		2017	21	21	12	2	4	4	0	
9	Ohloh	2015	3	0	0	0	1	3	0	78
		2017	21	21	12	2	4	4	0	

BSD (Breaking Self Descriptiveness), IMT (Ignoring Mime Type), IC (Ignoring Cache), ISC (Ignoring Status Code), FH (Forgetting Hypermedia), Tun(Tunneling), MC(Misusing Cookie)

Table 5.5Antipattern Detection for Stack Exchange and Facebook Version History

API Name	Version Number	BSD	FH	IMT	ISC	IC	MC	Tun
Stack Exchange	2.0	0	19	53	0	0	0	1
	2.1	0	24	53	0	0	0	1
	2.2	0	26	53	0	0	0	1
Facebook	2.7	67	29	8	2	0	0	0
	2.10	21	21	12	2	4	4	0

BSD (Breaking Self Descriptiveness), IMT (Ignoring Mime Type), IC (Ignoring Cache), ISC (Ignoring Status Code), FH (Forgetting Hypermedia), Tun(Tunneling), MC(Misusing Cookie).

Possible solution of comparison of antipattern evolution was to check the common method that were available in year 2015 to 2017. This information can also be checked from their online changelog reported in Table5.1. Let's consider the changes in service/id/friends that reports the antipattern tunneling in year 2015 and same for year 2017. But the information we got in response was different as compared to 2015.The two non-standardized header like x-fb-debug and x-fbrev was found in 2015 as compared to 4 non standardized header like x-fb-rev,x-fb-trace-id,facebook-api-version, x-fb-debug in 2017. This information also helped us to identify which type of parameter that each service providers used in its header field. We added these parameters to enrich the non-standardized header file for the correction of tunneling antipattern.

5.5.2 RQ2: How antipatterns are evolved?

We consider three case examples for this question that cover the changes in the evolution history of REST API.

Alchemy API: Alchemy API is constantly changing and also passed through the migration phase. The migration phase of Alchemy API is verified from their online trace history reported in Figure 5.2 and its traces are also available on SOFA website. Migration phase of alchemy API has effected the Mime type antipattern where its

instances were increased in May 2017 to July 2017 from 2 to 7. We also found the removal of Mime Type antipattern when we changed its request to the new address www.gateway-a.watsonplatform.net. This also proved that REST APIs undergo changes during its migration phase that effected the content type parameter for

response. However, this migration didn't effect on other antipatterns. We are also not able to find any instance of Misusing cookies over two moth continuous detection. Figure 5.6 shows the graphical analysis of entire evolution history of Alchemy API from May 2017 to July 2017.

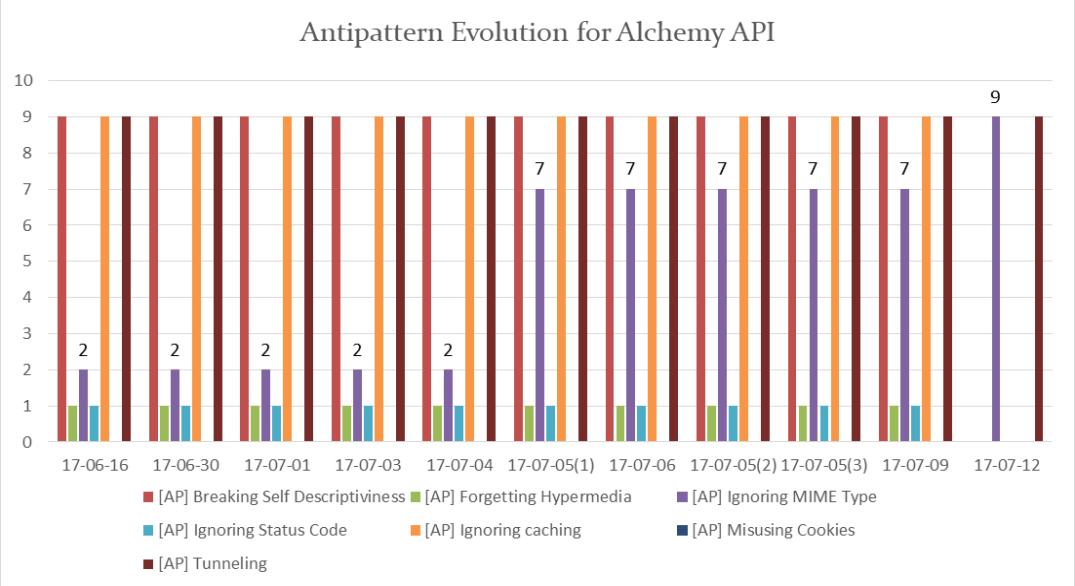


Figure 5.6 Antipatterns Evolution for Alchemy API

Bitly: is currently running its version 3.0 that follow OAuth 2.0 with SSL implementation. There were only few options available in their documentation regarding their API implementation. They just mentioned the status code description for 200,400,403,500 and 503. However, we implemented the complete status code description for all options from the forum [67] and checked the description of request and response for each REST API. There was no change for Ignoring Cache, Misusing Cookie and Ignoring Status Code. Tunneling antipattern was also shown 4 instances for its entire detection results. But we found little change for the forgetting hypermedia antipattern. The reason of change was the absence of" location' attribute in its response meta-data. The most effected services by this antipattern are /link /click,/shorten,/bitly_pro_domain,/user/tracking_domain,click/user/tracking_domain_list. This antipattern evolved in July and effected one more service name as/link/encoders_by_count. Manual validation proved that there was also a conflict regarding status code. Response body of this service shows "data","null" with status code 500 value. Figure 5.7 shows the graphical representation of Bitly antipattern evolution history.

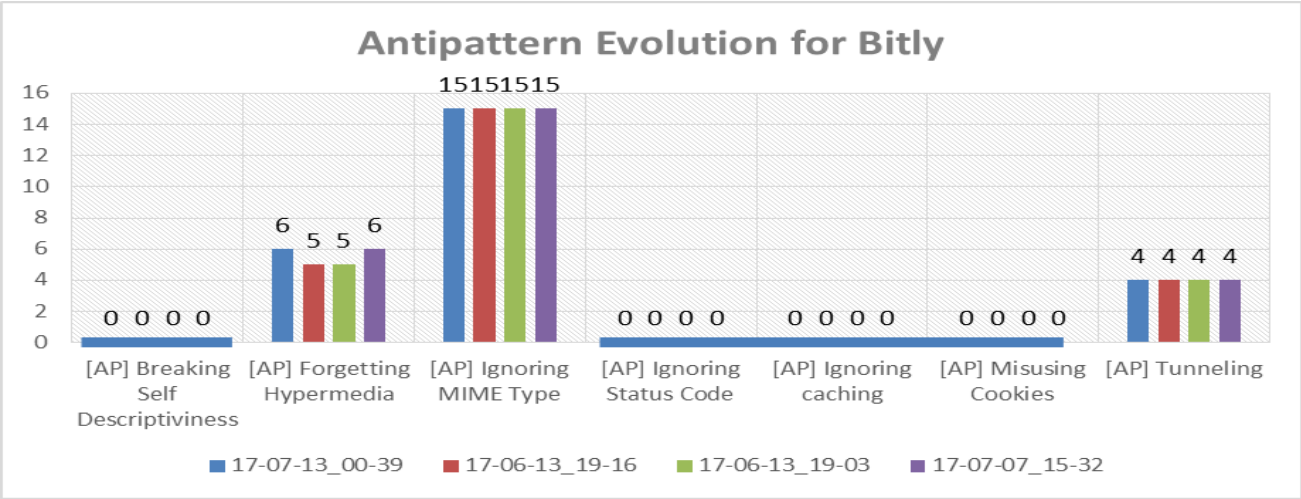
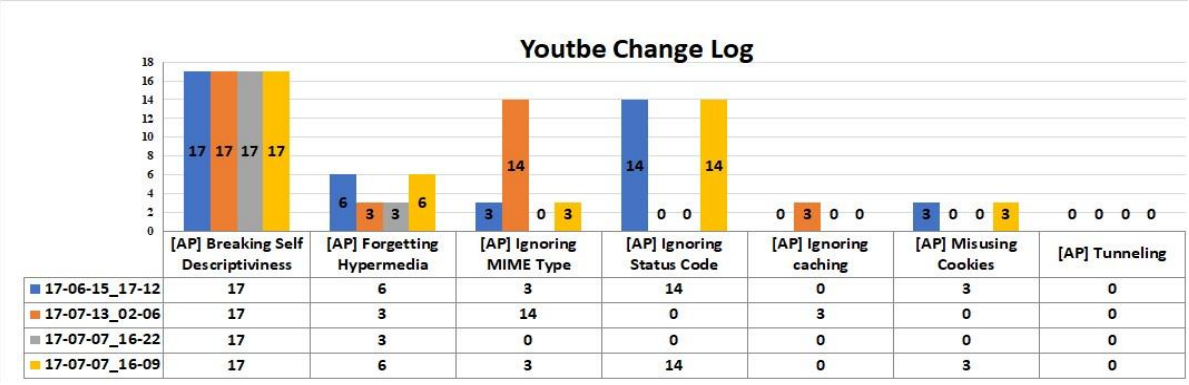


Figure 5.7 Antipatterns Evolution for Bitly API

YouTube:

YouTube has quite long change history and is constantly evolved over the past four years. We were unable to found any change on the instances of breaking self-descriptive antipattern and tunneling antipatterns from May 2017 to July 2017.

Ignoring Cache antipattern was evolved from one service to another. The major reason of this evolution that /video/rate was also not available while requesting for response. After checking the online changelog of YouTube we found that /video/rate is available till May 2017 but



	[AP] Breaking Self Descriptiviness	[AP] Forgetting Hypermedia	[AP] Ignoring MIME Type	[AP] Ignoring Status Code	[AP] Ignoring caching	[AP] Misusing Cookies	[AP] Tunneling
17-06-15_17-12	17	6	3	14	0	3	0
17-07-13_02-06	17	3	14	0	3	0	0
17-07-07_16-22	17	3	0	0	0	0	0
17-07-07_16-09	17	6	3	14	0	3	0

Figure 5.8 Antipatterns Evolution for YouTube API

some of its feature like recordingDetails.locationDescription, recordingDetails.location.latitude and recordingDetail.location.longitude was deprecated at June 1, 2017. This change also effected the response of this service. We also found non-standardized header description from activities, /playtimes, /guidecategoriesname asalt-svc. Ignoring mime type antipatterns were also found in /activities,/playtimes,/guidecategories where server required response in json but client ask in XML.

5.5.3 How antipatterns are removed?

We analyzed the complete trace history of each REST APIs for year 2005 to 2007 and investigated the changes in the instances of each antipatterns. We collected real time definition of each antipatterns correction by the REST API providers when subsequent antipatterns are removed by the REST API providers. We collected the correction approach of ignoring status code antipatterns from *Bitly and Drop Box*. Similarly, Charlihavery REST API reports the removal of Ignoring Mime type antipatterns. However, Tunneling through Get and Response antipatterns was not detected from Facebook and YouTube. The corrected antipatterns definition from traces helped to refine the correction definition of antipatterns reported in [56].

After analyzing each response and request for the correction of antipatterns, we implemented correction heuristics in SOCA-R (Service Oriented Correction for Antipatterns for REST APIs) to dynamically correct the antipatterns and improve the quality of REST API. This was also necessary that correction approach may not further introduced more antipatterns and improve the quality of the REST APIs by either increasing the total number of design patterns or removing the antipatterns from REST APIs. Table 5.3 reports the correction heuristics used by SOCAR that increase the instances of Entity-Linking, Response Caching and Contents Negotiation design patterns. Table 5. 6 reports the total number of antipatterns corrected for each REST API by using SOCA-R.

Table 5.6 Correction of Antipatterns by SOCA-R

Sr.No	API Name	Number of Services Analyzed	Total Antipatterns Detected	Total Antipatterns Corrected
1	StackExchange	53	72	72
2	Alchemy	9	31	31
3	ChalriHavery	12	12	12
4	Dropbox	17	45	45
5	ExternalLip	6	6	6
6	Music Graph	19	40	40
7	Ohloh	7	7	7
8	TeamViewer	19	14	14
9	Twitter	24	72	72
10	YouTube	14	36	36
11	Facebook	30	64	64

5.6 Threats to Validity

The goal of this study is to provide an approach for the improvement of REST API antipatterns, a generic methodology that will help industry to improve the quality of services.

5.6.1Construct Validity:

Threats related to construct Validity concern with the relationship between theory and observation, and checked the validity for the accuracy of the experiment we performed to address our research questions. We have conducted an online case study to know about the developer's perception about our correction approach used for the REST API antipatterns.

Objects:

For the purpose of the study, we performed evaluation of proposed research methodology for the correction of antipatterns for the REST APIs. First we checked the removal of these antipatterns from the tracelog collecting before and after correction. We provided correction rule to two independent researchers who were not part of this study.

Participant:

Ideally, a target population should be defined as a finite list of participants- i.e., the individual who can use the survey. Participant's involvement should be in valid numbers of the target population [218]. Non probabilistic sampling is the technique to define the sampling if target population is not clear. In this study, target population was all developers who were working in REST API and the students who used REST API as per the guidelines, it was impossible to define such list. Therefore, we selected the target population using convenience sampling. We invited participants by email and also shared the link of our questionnaire on famous developer's forums available at social media. We also posted our questionnaire in famous REST API provider's blogs like Facebook, YouTube,twitter,teamviewer e.t.c used in this study. 25 participants completed this study out of these 25 ,3 are graduated students and 22 are professional developers, provide a response rate close to 10 as expected [219]. Participants are volunteers and they did not receive any rewards for this survey. We explicitly told the professionals that we will maintain the anonymity of the results.

Study Design :

We prepared an online questionnaire and asked participants about their views regarding eight antipatterns and their refactoring operations needed to performed for the antipatterns correction reported in Table 5.7. The questionnaire was designed

based on the correction definition implemented in SOCA-R (Service Oriented Correction for Antipatterns for REST API) after collecting real time definition of traces collected by calling each service interface.

Table 5.7 Accuracy of SOCA-R(Service Oriented Correction for Antipatterns for REST API

Sr.No	Antipattern Name	Corrected Parameters	Agree	Not Agree	Corrected definition	P	R
1	Ignoring Mime Type	Content –type	23	2	json/xml	100	92
2	Ignoring status code	status code/status description	25	None	Replace wrong description/code with correct	100	100
3	Misusing Cookies	set- cookie	22	3	remove set cookie	100	88
4	Breaking Self Descriptiveness	Request header/Response Header field	24	1	HTTP information as per IETF	100	96
5	Forgetting Hypermedia	httpmethods,entity link,location	25	0	dynamically add location attribute and information if not available	100	100
6	Ignoring Cache	E-Tag,Cache Control	23	2	dynamically add unique E-tag for request/ enable cache	100	92
7	Tunneling	httpmethod,requesturi	23	2	HTTP information as per IETF	100	92

We informed the users about the antipatterns definition, correction definition used for each antipattern correction. Users were asked to check the correction definition and give their consent about the correction definition, if not then they may provide alternative solution for each antipattern. We also provided an example for each antipattern that how specific antipatterns corrected. Correction example contained the parameters needed to be corrected for REST API like status code, request, response, location attributes e.t.c . More detail of each refactoring operations is available in Table 5.3. The data collection survey is designed online and was available online for

two months. The survey used for the data collection is also reported in Appendix 2. The users from academia and industry fill the questionnaire and data were collected after survey was completed.

5.6.2 Internal Validity:

Threats to internal validity concern with the factors that may affect our results. When asking the opinions from users about the correction definition of antipatterns, we designed specific questions as per the correction definition of each antipattern. To mitigate this threat, we gave rights to users to negate our correction definition and may provide opinion that what was the best definition one can use for specific antipatterns. Developers may have right to negate our correction definition and give answers that can further be used for a correction definition if most of the community negate with the possible solution of antipatterns. Another threat to the validity, the solution of particular antipattern is already provided that may lead to an easy evaluation and subject to inconsistencies. We mitigate this threat by asking developers opinion about the way of correction for each specific antipattern.

5.5.3 External Validity

Threats to external validity concern with the generalizability of SOCA-R findings. Interims of accuracy, we corrected all instances of reported antipatterns. We implemented real time correction definition of famous REST APIs like YouTube, Facebook, StackExchange after collecting their tracelogs from 2014 to 2017 and then use corrected parameters in SOCA-R for the correction of antipatterns instances. Precision of our tool is calculated as

$$\textbf{Precision} = TP/TP+FP$$

where TP is True Positive and FP is False Positive. We calculated the precision of SCOA-R as ratio of correctly predicted observation over total number of observations. Here correctly predicted observations are all reported antipatterns by SODAR and then we developed correction algorithm in SOCA-R to correct all reported predicted antipatterns in major REST APIs tested. Recall of the SOCA-R was calculated as Recall is the ratio of correct prediction over total observation in actual class.

$$\textbf{Recall} = TP/TP+FN$$

We checked the developers and users opinion about the correction definition. For reporting the recall, we calculated the number of user's opinions who negated with

our correction definition. Precision of SOCA-R was 100 percent whereas recall was 94 percent. Table 5.7 reports the accuracy of each antipattern correction by SOCA-R.

5.6 Conclusion

The analysis of evolution of REST APIs for the correction of antipatterns helped us to know about the overall strategy of REST API designs by famous REST API providers. REST API changeproness cannot be monitored directly due to the unavailability of trace history by famous REST API providers. There was also lack of information regarding number of errors reported for each REST API over specific period of time. Every REST API providers used their own guidelines for designing REST API and no mechanism exist that force REST API providers to report the number of changes for each REST API services. YouTube started its change log for version history first time in 2013. There is no online database that provide the number of errors or user responses for each specific change for specific web service.

There are number of approaches that report the REST APIs design schema and information regarding antipatterns and design patterns, however no information is available for the correction of errors or any empirical study that report the solution of changes for REST API for one version or for a specific period of time. There is no publically available dataset that provide complete information for REST APIs like trace log, effect of evolution on service providers when REST API undergo various changes as we noticed in the case of Alchemy when they shifted from Alchemy to watsonplatform.net. Most of the companies like twitter or dropbox even not provide changelog to analyses the changes for each version. We developed SOCA-R for the correction for REST antipatterns with precision of 100 percent and recall 94 percent. We validated our results by conducting an online survey and asked opinion from industry as well as academia that are working in the area of REST API development. We reported recall after collecting the results of survey. Most of the user accept our correction approach. There was only 25 percent negation for antipattern that is Ignoring Mime Type because now a day's industry more relies on json as compare to xml. However, it is recommended that use different representation of Mime type. The industry also provided its views for correction of misusing cookie by setting set cookie. The respondents of the survey focused to avoid set cookie field instead of remove tokens or key in set-cookie field. Using an extended SOFA framework

(Service Oriented Framework for Antipatterns), we analyzed 115 methods and mined their trace history for the correction heuristic from 2014 to 2017.

Chapter 6 Correction of Linguistic Antipatterns for RESTful API

6.1 Introduction

Representational State Transfer (REST) architectural style is a popular style widely used by industry introduced by Fielder et al.[6] . REST helps to publish a set of multiple interconnected resources for client to discover hyperlinks and interact with each other by using their uniform interface. Interconnected resources imply that clients can achieve multiple targets without creating problem for application state. The client communicates with multiple REST APIs using basic set of Hyper Text Transfer Protocol (HTTP) with the combination of URI and MIME types that helps to maintain interoperability and simplicity for various platforms [6]. Service-Oriented Architecture (SOA) has provided new direction for the software developers to develop fully distributed and customizable applications used by large software industry like Facebook, Twitter, YouTube, Dropbox in the form of RESTFUL APIs.

RESTful APIs must use coherent naming rules that may attract client developers as compare to poorly design naming rules. The usage of best practices for RESTful APIs design also improve the design and development time for web based applications. The improper use of linguistic relations for parameters, resources and services are crucial for designing RESTful APIs that may introduce antipatterns as compared to the good design RESTful resources call as design patterns. Quality of design and development of the RESTful APIs can be improved by adopting good design practices [206].

In the context of semantic and syntactic knowledge about RESTful API poor design practices, a recent study by Palma et al. [57] highlighted the detection of REST linguistic antipatterns from12 widely used REST APIs with a tool support DOLAR[57] and SARA[58]. The researchers focused on the analysis of poor practices [2][5[6] in URI nodes or in cloud based API[7][8]. Consider the example of www.illustration.com/mathmatical/essay where math and essay URI nodes have no sematic context and identified as Contextless Resource name [208] as compared to www.illustration.com/mathmatical/trignomatry identified as Contextualized Resource Name [208]. Contextualized resource name helped developers for better understanding and interaction with server. Researchers also highlighted the issues in REST cloud computing lexicon with a tool support CLOUDLEX [164]. The poor design practices are also validated by another study after examining the REST API of Google Cloud Platform, OpenStack, and Open Cloud.

The researchers studied the improvement in source code lexicon for OOSE(Object Oriented Software Engineering) but we are not able to identify any tool support for the improvement in RESTful API linguistic antipatterns correction. This study will bridge the gap from the detection of REST API linguistic antipatterns towards the correction of linguistic antipatterns after extending the SOFA architecture by introducing new linguistic antipatterns correction handler.

6.2 Related Work

Service based systems are continually evolved to meet the rapidly changing user requirements. There is a need to improve the QOS (Quality of Services) for the service based systems by detecting and correcting antipatterns. This study only aims to focus QOS issues for Web Services specifically for RESTful APIs. There are different approaches available for antipatterns correction for Webs services as well as for OOSE. This section highlights the importance of lexical analysis for the improvement in program understandability, maintainability that also helps to remove the antipatterns.

6.2.1 Analysis for Web services

There are number of contributions available in the field of the lexical analysis for the improvement in web services availability and readability for REST API developers.

Design principles and RESTful APIs

Masse et al. [206] in his book defined six rules for RESTfull APIs design related to URI , request, responses and representation of hypermedia. These design principles are further used by another study for the compilation of 73 best practices. These best practices are further investigated for Google cloud, Open Stack and OCCI1.2 [120]. The results of this case study show that major REST APIs providers do not adopt good practices completely [120]. Another study also highlighted the lexical issues in RESTfull APIs after parsing their documentation and extracting URI, child nodes for RESTfull web services. Lexical analysis of RESTful APIs URI showed that majority of the REST cloud computing APIs use nouns and they do not share common lexicon globally as no commonality is found between 352 different terms collected from Google Cloud Platform, OpenStack, and Open Cloud Computing Interface. Palma et al. [66] also discussed a new antipattern in his study named as *Pertinent versus Nonpertinent Documentation that check the URI* with respect to the documentation of

15 widely used REST API with a tool support SARA. The results of the study showed that 72% URIs are consistent with their documentation as compared to 28 percent that are not persistent with their documentation.

Rodríguez et al. [170] highlighted good and bad design principles for android REST APIs. The dataset used for the evaluation related to the HTTP data logs is collected after examining the internet traffic. Zhou et al. [153] also reported the issues that will help to fix REST design problems for the REST services in existing Northbound networking APIs specifically for a Software Defined Network. Zhou et al. [165] also guided the users regarding design of a REST Northbound API specifically for the OpenStack . Maleshkova et al.[208] investigated the set of 220 Web APIs publicly-available, including RPC, REST, and hybrid. The characteristics under analysis were general information, types of Web APIs, input parameters, output formats, invocation details, and complementary documentation. This work helped users to know about the development and usage of REST APIs.

Antipatterns detection for web services

There are a number of antipatterns detection approaches available for the Web Services. A catalog of Web service Antipatterns is presented in [170]. These antipatterns also addressed previously identified WSDL antipatterns issues like name issues [47] and data type definition issues [70]. Moreover, it is also very difficult to assessed the quality of Web services that were affected by non-representative names as well as unclear documentation [52]. Different approaches discussed antipatterns detection for WSDL document [48,49,51,52,53,54] that assists developers for developing, discovering, publishing and consuming web services. Palma et al.[47]addressed the detection of antipatterns from SOAP based services . SOAP antipatterns were also detected by using PE-A(Parallel Evolutionary Algorithm) based on genetic algorithm [54]. Researchers also used the machine learning and genetic programming algorithm for the detection of antipatterns and design defects among SOAP web service [212][52].

There are number of approaches that detect antipatterns based on the industry code-first and contract first approach [78]. Antipatterns detection from some of the above mentioned researches focused towards static analysis of the WSDL document and then suggested refactoring approaches needed to remove these antipatterns. But the

above mentioned researches don't provided any technique with tool support for the correction of linguistic antipatterns.

6.2.2Analyses for OOSE.

Lexicon analysis for Source code

Researchers claimed that bad coding practices related to the fault proness [84] and affected the understandability of source code [213]. Researchers also applied ontology that used to extract relations between source code and improve the concept understandability [214]. This technique helped developers to choose right identifiers consistent with the concept of the overall system [215]. They also suggested to use ranking techniques for the replacement of identifiers not following standardized quality criteria [215]. The tool name COCONUT by De Lucia et al. [219] ensures consistency between high level artifacts and source code lexicon. The evaluation of source code quality was based on the results of textual similarity between source code lexicon and high level source code artifacts.

There were multiple studies that highlighted the importance of identifiers [220], [209], on maintainability program comprehension and quality of the source code. Identifiers were used for the recovery of traceability links, [222][223], understandability and maintainability [224]and conceptual cohesion and coupling [41][46]. Verdancaet al. [225] also highlighted the importance synonyms, antonyms and hyponyms for the understandability of the program by conducting a case study and with a tool support. However, above all mentioned approaches inspired us for the improvement in linguistic antipatterns for RESTful APIs but cannot be applied directly as services provide interface as compare to OOSE where source code of application is available.

Correction of Linguistic Antipatterns for OOSE

Weissgerber and Diehl [226] proposed a signature based methodology for the identification of refactoring. The Signature based approach started with the use of preprocessing data for collecting information from version control system. Moreover, they detected multiple classes, fields, interfaces in different version control system. The resultant information from different version control systems was used for refactoring. The tool named as REF-FINDER helped to detect method renaming after

comparing similarity between body of two methods [227].The RENAMING DETECTOR proposed by Malphol also helped to detect identifiers renaming after matching their declaration in two different versions [229]. Precision was reported 100 percent for 77 analyzed systems [229]. The source code evolution was analyzed with the help of AST by Neamtiu et al. [209] that summarized changes in two systems with the help of program written in C. The objected oriented refactoring was also identified with the help of change count metrics calculated from source code from two versions of Smalltalk program [209].

Fluri et al. [211] proposed a tree differencing algorithm, CHANGEDISTILLER, for extracting the changes from two consecutive versions of Java files. Fluri et al. [211] proposed a tree differencing algorithm, CHANGEDISTILLER, for extracting the changes from two consecutive versions of Java files. Renaming is a type of change that CHANGEDISTILLER can detected together with many others. The algorithm compared the ASTs of the files and computed the edit operations to transform the AST of the old version of a file to the AST of the new version of the same file. The association rule discovery [58] was also used for the detection of change coupling .The renaming of an entity considered as “essential “change while their updating considered as “non-essential”. They proposed tool named as DIFFCAT which based on the approach of [211] and tested for dnsjava and JBoss.

The above mentioned studies reported for detection of OOSE and correction of web services antipatterns for SOAP based web services. Some recent studies also reported the evolution of antipatterns for web services. As per best of our knowledge we are not able to find approach for the correction of linguistic antipatterns for RESTful APIs.

6.3 Correction of Linguistic Antipatterns in REST APIs.

Correction Approach for linguistic antipatterns for REST APIs is based on the SOFA (Service Oriented Framework for Analysis) with a tool support DOLAR(Detection of Linguistic Antipatterns in REST) [57]. COLAR(Correction of Linguistic Antipatterns in REST) used approach of DOLAR with an extension of correction handler for REST API for solving linguistic antipatterns problems. COLAR approach is based on the following steps as shown in Figure 6.1.

1. The use of DOLAR for linguistic antipatterns detection for REST API. This step is used to extract the traces of RESTful APIs for the analysis of complete URI used for request of RESTfull web services.
2. The detailed analyses of the definition of linguistic antipatterns is collected from the literature [206] [207]. The violation of design principles for RESTfull APIs also helped us to report the correction approach [206] [207].
3. Generate tokenization of URI parameters to perform refactoring for the correction of antipatterns.
4. Implementation of correction algorithm after collecting definition from literature and develop a correction handler for SOFA suite.
5. Automatic correction of linguistic antipatterns is added in SOFA suite.The complete methodology of COLAR tool is given below in Figure 6. 1.

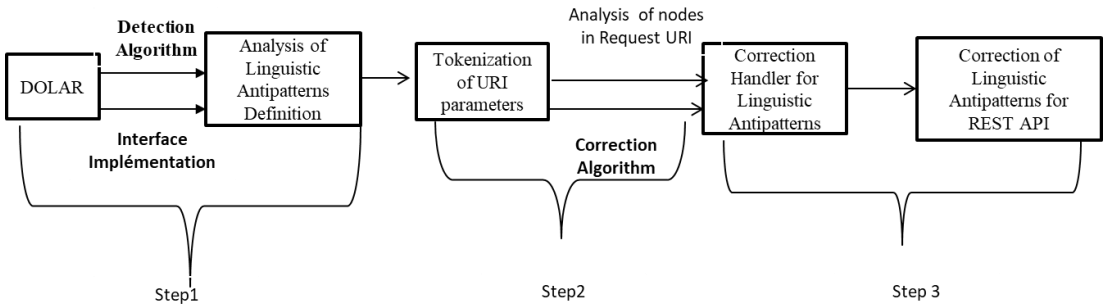


Figure 6.1 Correction of Linguistic Antipatterns

6.3.1 Definition Analysis for RESTful Linguistic Patterns and Antipatterns

Verbless design pattern vs. CRUDY URIs anti pattern

The use of GET, PUT ,POST or DELETE should be used in URI provided verbless design pattern as compared to the CRUDY(create ,update ,delete) terms in CRUDY URI antipatterns [206][207] .

Example:

POST as https://www.abcexample.com/create/list/customer?id=123 is a CRUDy URIs antipattern because it contains a CRUDy term 'delete' while creating customer having id 123. The use of Verbless design pattern is to avoid CRUDY terms and the correct URI will be as *POST https://www.abcexample.com/list/customer?id=123* as this is used HTTP POST request without any verb.

Correction Algorithm:

```
   Input:  node ,http method
   Output:  Verbless Design Pattern
 1 node⟵CRUDY words ;
 2 httpmethod⟵CRUDY words ;
 3 //if node contains multiple words check each word sperately
 4 if node "contains" CRUDY words then
 5 |  Replace CRUDY word with " " ;
 6 end
 7 correcthhtpmethod= getAppropriateMethod(crudyWord)
 8 if correcthttpmethod != httpmethod then
 9 |  Replace http method with Correct http method
10 end
11  " CRUDY URI Antipatterns removed "
12 "Verbless Design pattern detected"
```

Figure 6.2 Correction Heuristics for CRUDY Antipatterns

Figure6. 2 shows the correction algorithm used for the correction of CRUDY URI antipattern. We extended the DOLLAR [57] by implementing correction algorithm that provided the missing parameters and correct the URI dynamically. We reported RESTful API URI as CRUDY URI if it contains the CRUDY (create, update, delete) words in the URI nodes as well as http method implemented by this URI. Each node was checked for CRUDY words and then COLAR removed these antipatterns. After removal of CRUDY antipatterns the instances of Verbless design pattern detected as URI now fullfill the criteria of Verbless design pattern definition.

Hierarchical vs. Non-hierarchical Nodes

Description: The URI is the combination of multiple nodes and each node must be semantically related to its neighbor nodes. Non-Hierarchical nodes antipattern represent one irrelevant node among all URI.

Example:

https://www.abcexample.com/Department/fee/ faculty is an example of a Non-hierarchical Nodes antipattern since 'department', 'fee', and 'faculty' have not a hierarchical relationship.

The URI *https://www.example.com/university/faculty/professors/* is a Hierarchical Nodes pattern since 'university', 'faculty', and 'professors' are in a hierarchical relationship.

Correction algorithm

```
Input: nodesinURI
Output: Hierarchical Design Pattern
nodesinURI⟵nodeA,nodeB ;
 int count = nodes.size();
if (nodes.size() >= 2) then
    for (int i = 0; i< nodes.size() - 1; i++)
    ArrayList<String> nodeA = nodes.get(i);
    // Get first node and second node for comparing
    ArrayList<String> nodeB = nodes.get(i+1);
    // checks for hierarchy String relationType =
     HierarchicalMetrics.reversedHierarchy(nodeA, nodeB);
     System.out.println("Relation type : " + relationType);
    if (!relationType.equals("")) then
        String path2 = correctedUriWithoutParam.replace(".json",
         "");
        // Remove extension from URI to get total number of nodes
        originalNodesArray = path2.split("/");
        originalNodesSize = originalNodesArray.length;
        index = originalNodesSize-count;
        // String "abcd" used for nodes swapping purpose
        correctedUriWithoutParam=correctedUriWithoutParam.replace
        (originalNodesArray[index+1], "abcd");
        correctedUriWithoutParam=correctedUriWithoutParam.replace
        (originalNodesArray[index], originalNodesArray[index+1]);
        correctedUriWithoutParam=correctedUriWithoutParam.replace
        ("abcd", originalNodesArray[index]);
        CorrectedURI after replacing nodes
    end
end
 "NON Hierarchical Nodes Antipatterns removed "
"Hierarchical nodes Design pattern detected"
```

Figure 6.3 Correction Heuristics for Hierarchical Nodes Antipatterns

Figure 6.3 shows the implementation algorithm used for the Non-Hierarchical nodes antipatterns correction. This antipattern focused on the formation of nodes and their relevancy with their child nodes. The nodes are collected and checked after removing extensions from the URI. We used CORENLP for the checking the semantic of the nodes and then replace their position after swapping. This technique helped us to add

most relevant parent node first and then child nodes at the last. Nodes must be singular after swapping and fulfill the definition of Hierarchical nodes design pattern.

Tidy vs. Amorphous URIs

Description: REST is an architectural style that promote easy readability for resource URIs. A Tidy URI is REST linguistic design pattern that promotes the use of lowercase letter for resource naming without the use of extension, underscore and trailing slashes.In contrast, Amorphous URI antipatterns detected when URI used symbols and capital letter that created problem for URI readability.

Example:

*https://www.abcexample.com/university/faculty/profile/biodata*is Tidy URI as compared to*https://www.abcexample.com/University/Faculty/pic.jpg* is an Amorphous URI antipatterns as use of uppercase letters and extension is appeared in URI.

Correction algorithm :

```
   Input:  URIparameters
   Output:  Hierarchical Nodes Design Pattern
 1 URIparameters⟵Hyphenated URI,UpperCase URI,Trailing Slash
    URI,Extension URI ;
 2 if URIparamters "contains" Hyphen then
 3 |  correctedURI= path.replace(_ ,-) else
 4 |  if URIparameters contains UpperCase URI then
 5 |  |  correctedURI= correctedURI.toLowerCase
 6 |  |  if URIparameters contains Trilingslash URI then
 7 |  |  |  correctedURI= correctedURI.remove (/,\\)
 8 |  |  |  if URIparameters contains Extension URI then
 9 |  |  |  |  if correctedURI.contains(extension.toLowercase() OR
   |  |  |  |   extension.touppercase()) then
10 |  |  |  |  |  correctedURI=correctedURI.remove(extension)
11 |  |  |  |  end
12 |  |  |  end
13 |  |  end
14 |  end
15 end
16  " Amorphous Antipatterns removed "
17 "Tidy Design pattern detected"                                Activate
```

Figure 6.4Correction Heuristics for Amorphous Antipattern

Amorphous antipatterns removal consists on several steps.

1. It checked the hyphen in URI as use of hyphen in URI nodes was considered as bad design practice for designing RESTful API [206] [207]. The removal of this antipattern based on the removal of hyphenated URI with dash '-'sign. Moreover, there was a need to remove the uppercase letter in the URI to make URI more readable by client.
2. Use of final trailing slash is not a good choice for designing RESTful resources. The removal of this antipatterns is based on the parsing of final or last URI node and then removing '/' from URI nodes.
3. The Extensions and upper case letters should be removed from the RESTful resources.

The implemented correction algorithm must fulfill above all criteria after calling the interface of each RESTful API and then removing all above issues to make tidy design pattern.

Singularised vs. Pluralised Nodes

Description: URIs must use singular/plural nouns for the representation of resources for all APIs. When client requested any information using PUT/DELETE, the last node that acquire URI must be singular. The last node of URI should be plural in case requested node contain POST request. The Pluralised antipattern does not effect GET request. Therefore, it is required to use Singular name for PUT/DELETE to avoid this antipattern.

Example:

For POST operations

https://www.abcexample.com/university/faculty/profiles is Pluralised node antipattern

For DELETE operation

https://www.abcexample.com/university/faculty/messages

Correction Algorithm

```
   Input: httpMethod
   Output: Singluarized Design Pattern
1  httpMethod←—PUT,POST,DELETE ;
2  if httpMethod "contains" PUT or DELETE then
3  |  String singularNode = inflector.singularize(word);
4  |  // Used inflector liberary to convert plural node into singular
5  |  correctedtedURIWithParams = URIForProcessing.replace(word,
   |   singularNode);
6  |  URIForProcessing = correctedtedURIWithParams;
7  |  if httpMethod contains POST then
8  |  |  String singularNode = inflector.singularize(word);
9  |  |  // Used inflector liberary to convert plural node into singular
   |  |   correctedtedURIWithParams =
   |  |   URIForProcessing.replace(word, plularNode);
   |  |   URIForProcessing = correctedtedURIWithParams;
10 |  end
11 end
12  " Plurised Antipatterns removed "
13 "Singularized Design pattern detected"
```

Figure 6.5 Correction Heuristics for Pluralised Antipatterns

The correction of Pluralised antipattern was based on the inflector library that convert plural node into singular node. The refactoring of *Pluralised antipattern* first checked

the client request. The inflector library helped us to remove the plural words in to singular words in case of PUT and DELETE. The removal of words based on the appearance of plural words in the last node of the URI. Similarly, use of singular name is also prohibited in case of POST request from client. The implementation of correction algorithm first checked the method used for the request and then implemented the strategy used for the correction of Pluralised antipattern.

6.3.2 Implementation of Correction Algorithms

Correction algorithms are implemented in java by extending SOFA framework and adding RESFTfull Linguistic antipatterns correction handler. Request are analyzed using trace history as well as using WORDNET[11] and CORENLP dictionary after extracting tokens for each request of RESTful API. Refactoring operations are performed for each antipattern separated after transforming rules manually implemented in java.

The COLAR(Correction of Linguistic Antipatterns for RESTfulAPI) tool used correction algorithm after extending the SOFA(Service Oriented Framework for Antipatterns). The algorithmic rules were translated in form of refactoring operations performed for each antipatterns .

For the correction algorithm we mainly rely on DOLAR approach that used *FraSCAti IntentHandler* for extracting runtime parameters for complete request and response. The correction algorithm provided complete trace log for a correction of each antipatterns.

6.4Analysis of Results

This section reports the detailed results of correction of linguistic antipatterns from RESTful API along with the accuracy of the correction algorithm. Table 6.1 reports the total number of services analyzed for each RESTful API under analysis.

Table 6.1 Online documentation of Services Analysed

Sr.No	RESTful API name	Online Documentation	No of APIs under analysis
1	YouTube	https://developers.google.com/youtube/documentation	17
2	Facebook	https://developers.facebook.com/docs	27
3	Alchemy	https://www.ibm.com/watson/developercloud/doc/index.html	9
4	Twitter	https://dev.twitter.com/en/docs	25
5	Bitly	https://dev.bitly.com/documentation.html	
6	CharlieHarvey	https://charlieharvey.org.uk/page/api_docs_prelaunch	12
7	Externalip	https://Eden.openovate.com/documentation/social/zappos	6
8	StackExchange	https://www.sackoverflow.com/documentation	52
9	Ohloh	https://www.openhub.net	7
10	Dropbox	https://dropbox.github.io/dropbox-api-v2-explorer	17
11	Zappos	https://www.programmableweb.com/api/zappos	9
12	MusicGraph	https://developer.musicgraph.com	19

Total 200 services are analyzed for the analysis of the correction of linguistic antipatterns. The results of the correction of linguistic antipatterns are reported in Table6. 2.

Table 6.2 Results of correction of Linguistic Antipatterns

Sr.No	API Name	Services Analyzed	AURI	CURI	PN	NHN	Total Detection Total Corrections	% Corrected
1	Alchemy(AP)	9	8	0	0	9	17	50%
	APC		8	0	0	0	8	
2	Music Graph	19	19	0	0	19	38	50%
	APC		19	0	0	0	19	
3	Bitly	20	15	0	0	10	25	60%
	APC		15	0	0	0	15	
4	DropBox	17	14	6	8	15	43	65%
	APC		14	6	8	0	28	
5	Twitter	25	25	5	1	25	56	70&
	APC		25	5	1	0	31	
6	YouTube	17	10	5	1	25	41	39%
	APC		10	5	1	0	16	
7	CharliHavery	12	0	0	0	9	9	0%
	APC		0	0	0	0	0	
8	Zappos	9	9	1	0	0	10	100%
	APC		9	1	0	0	10	
9	Ohloh	7	7	0	0	5	12	58%
	APC		7	0	0	0	7	
10	Facebook	30	29	0	0	3	32	
	APC		29	0	0	0	29	90%
						Average Corrected		58%

AURI(Amorphous URI) , FHM(Pluralised Nodes),CURI(Crudy URI),NHN(Non Hierarchical nodes) APC(Antipattern correction)

6.4 Discussion on Results

Amorphous URI antipatterns are reported mostly in all RESTful APIs excluding Bitly and CharlieHarvery . The results of Linguistic antipatterns detection using DOLAR not varies also in year 2018. The instances of antipatterns are not increased or decreased for Linguistic antipattern detection. The reason of getting same results are that antipatterns detection perform on the request resource URI that are not vary and this can also be confirmed from their online documentation. Some of the RESTful services are not available in 2018 as analyzed for Facebook. Therefore, we selected 30 methods from Facebook and perform their detection and correction to check the number of antipatterns reported by DOLAR and corrected by COLAR. Facebook is suffering from Amorphous antipatterns mostly as compared to other linguistic antipatterns as we detected 29 instances for Amorphous antipatterns and we corrected all 29 instances. However, we are not able to correct the single instances of Nonhierarchical nodes. We also found Nonhierarchical nodes in most of the RESTful APIs and we also didn't find the relation between their nodes to justify the Hierarchical node antipatterns. One of the reason to not correct the Non-Hierarchical nodes antipatterns as most of the APIs have single nodes. We successfully removed some antipattern from twitter API completely. We also found Pluralized nodes antipattern in YouTube API where POST method is called but resultant node is singular as reported in Figure 6.7.

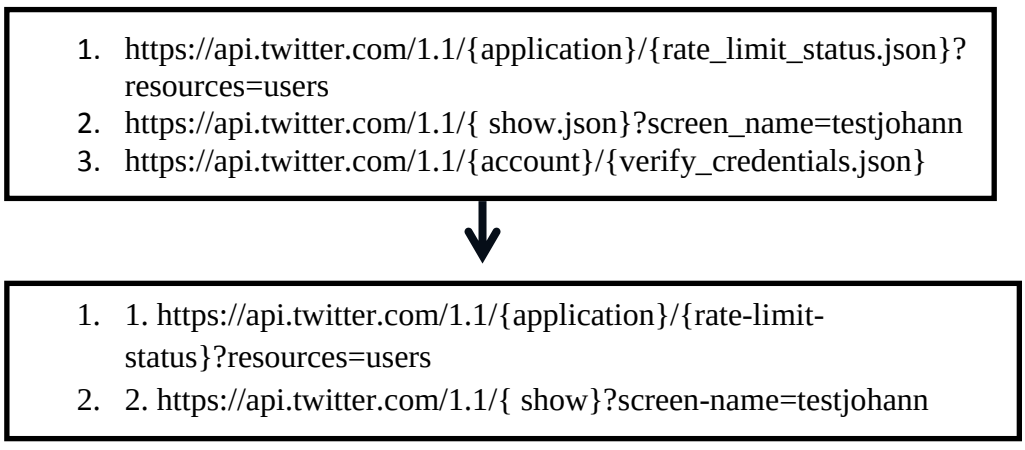


Figure 6.6 Traces of Antipattern Correction for Twitter

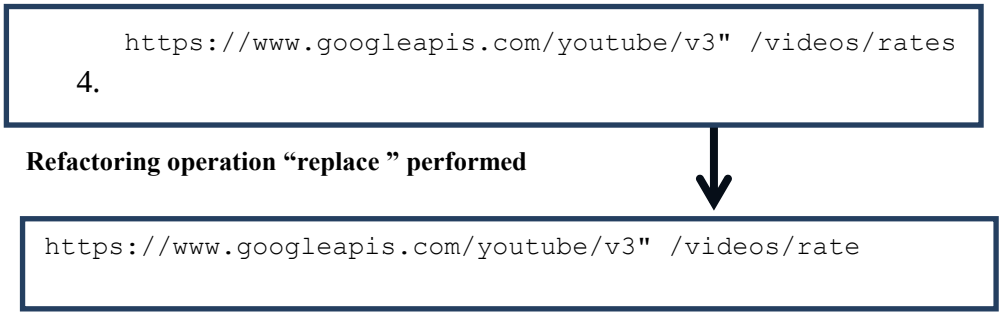


Figure 6.7.Traces of antipattern correction for YouTube

Amorphous antipattern is reported in all APIs as APIs providers don’t follow the guidelines for RESTful API design. Twitter reported 25 instances and Stack Exchange 28 instances for Amorphous Antipattern detection that are successfully removed by COLAR. The Amorphous antipattern depends on multiple conditions and most of the APIs violate the principal of good design. The complete traces are available for further analysis of URI.

6.5 Validity:

We examined total 200 methods for 5 antipatterns and it was quite difficult task to validate answers of 1000 questions. Therefore, we randomly selected some requested URIs from the set of 10REST APIs. The findings of the previous studies showed that Facebook and YouTube is well designed but as APIs are under constant evolution so we decided to select the methods from all APIs under analysis given priority to Facebook, YouTube, Twitter in terms of selecting maximum request. We manually validated the results of correction by three professionals to investigate the true positives, false positives and false negatives. We provided antipatterns definition to professionals to measure the accuracy of COLAR tool. The precision and recall is used as an accuracy measures.

Precision is reported as percentage between the true corrected antipatterns and all corrected antipatterns. Recall is reported as the percentage between the true corrected antipatterns and all available true antipatterns among RESTful API URI. The average precision of COLAR tool is 79 % and average recall is 75%. We are not able to

correct the non-hierarchical nodes antipatterns, that decrease the recall and precision of COLAR tool.

Table 6.3 Relative Accuracy Measures of COLAR Tool

Sr.No	RESTful API name	Method Tested	P	TP	Validated	Precision	Recall
1	YouTube	8	6	5	5	83%	100
2	Facebook	8	6	4	3	50%	75%
3	Alchemy	2	2	2	2	100%	100%
4	Twitter	2	1	1	1	100%	100%
5	Bitly	1	1	1	1	100%	100%
6	CharlieHarvey	1	1	1	1	100%	100%
7	Externalip	1	1	1	1	100%	100%
8	StackExchange	7	7	6	5	71%	83%
9	Ohloh	1	1	1	1	100%	100%
10	Dropbox	2	2	2	1	50%	50%
11	Zappos	2	2	2	1	50%	50%
12	MusicGraph	2	2	2	1	50%	50%
						Average 79 %	**Average 75%**

Threats to Validity

We performed correction on 12 widely used REST APIs by calling their 200 methods to minimize the threat to external validity. For lexical analyses of URI nodes we used the WORDNet for lexical as well as semantic analyses. Threat *to construct validity* deals with the association between theory and observation. We try to mitigate this rule by performing manual validation and involving post graduate students who were not involved in the implementation of correction algorithm and helped us to calculate the

precision and recall. We implemented the correction definitions after complete review of the definition of antipatterns and design patterns and then transform manual correction rules in java to perform correction. We tried to coup with *external validity* by selecting suitable representation of RESTful APIs. However, correction rules are based on the DOLAR tool architecture that perform detection and already have very low precision for Nonhierarchical nodes antipattern. The results of correction can vary based on the NLP algorithms used for checking the similarity between nodes as every algorithm has specific rule of implementation that may vary the results of accuracy. Threat to *Internal* validity concerns with the factor that may affect our results. The one reason of this type of threat could be the possible reason of subjectiveness due to manual implementation of rules. We define the correction algorithm in detail to avoid any biasness.

6.6 Conclusion

RESTful APIs are gaining popularity day by day and need constant maintenance as there are number of versions rapidly introduced by RESTful APIs provider's over the last few years. There is a need to follow the design principles while updating and maintaining RESTful APIs. This chapter presented a novel approach COLAR(Correction of Linguistic Antipatterns for RESTful APIs) tool for the correction of their linguistic antipatterns, an extension to the DOLAR tool. SOFA framework is extended by adding correction definition of linguistic antipatterns with an average precision of 79%. We partially validated the results of COLAR on the representative sample of RESTful APIs that shows average recall 75%.We found that most of the RESTful APIs are suffered from Amorphous URI antipattern. COLAR successfully corrected all instances of Amorphous URI antipattern.

Chapter 7 Conclusion

7.1 Conclusion

The Service Oriented Architecture is widely used by industry having multiple technologies like SOAP,REST and SCA. SOAP and REST services are widely used by industry like Facebook, Twitter, Google, LinkedIn e.t.c. SOA technologies are subject to functional as well as non-functional requirements that force service developers to meet the end user requirements in a short period of time. These rapidly changing requirements sometime lead towards the service design issues also called as antipatterns. These antipatterns effect the evolution of services, degrade the performance and raise different maintenance issues. The detection of antipatterns is important to improve the quality of services. There are different technologies used by industry to implement the services. The SOAP based services are implemented using code-first as well as contract-first approaches. There was a need to propose a generic approach for the detection of SOAP antipatterns that can be used by industry to detect antipatterns at interface level as well as code-first level. The current literature still so far did not report the correction of antipatterns as well as evolution of antipatterns for RESTful APIs. We tried to bridge the gap in research with a tool support SWAD for SOAP web services and SOCA-R along with COLAR for REST web services.

The analysis of primary studies by using systematic literature review helped us to identify the gap in literature and we extracted the data of 77 primary studies to investigate the literature thoroughly for the selection of problem domain. The detection of SOAP antipatterns are proposed as per the industry standards and we used Sparx System Enterprise Architecture to modeled for SOAP web services using SOAML.

We have identified problems in literature as 1) generic approach for the SOAP antipatterns detection based on industry techniques,2) evolution of antipatterns for various RESTful services over the last 2 years, 3) propose a correction approach for RESTful APIs to the quality of services and (4) the correction approach for the improvement in URI design of RESTful APIs. We proposed a solution for three research questions.

RQ1 **– What type of unique and lightweight approach that can be used by academia and industry for the detection of antipatterns for SOAP web services?**

SWAD (Specification of Web service Antipatterns Detection) is proposed that can be used by the industry for the detection of antipatterns at interface level and at implementation level for SOAP web services.This tool used variable threshold adaptation implemented via GUI that give rights to the developers to set threshold as per their choice for the detection of different antipatterns.

RQ2- How antipatterns are evolved for REST APIs?

The evolution of antipatterns are covered over year 2014 to year 2017 after calling service interface and mining trace history to check the attributes properties used by RESTful API providers for the correction of antipatterns. The evolution results can be confirmed from the trace history.

RQ3-How antipatterns for REST services are corrected?

The SOCA-R and COLAR tool is proposed for the correction of RESTful APIs antipatterns. The SOCA-R is evaluated by industry and academia and having precision and recall 94%. The COLAR tool is proposed that can assist RESTful API designers for the correction of RESTful API linguistic errors.

The SOCA-R is the only tool available still so far that dynamically correct the RESTful API with the support of SOFA framework proposed by Palma et al. [57]. The SOCA-R is further evaluated by industry and academia.

7.2 Implications of Research

We advise researchers to pay more attention to linguistic smells that are gaining popularity in the last five years. Moreover, research on the correction of lexical smells requires further investigation. Inter-smells relationship for lexical smells and performance evaluations of lexical design patterns *vs.* antipatterns are yet to be studied. Refactoring is a major area that is well researched for OO smells but not yet for SCA and REST smells due to their complex nature.

Developers' maintenance effort for smells in SOA systems is still not addressed in the recent studies. We were also unable to find any antipattern detected in Java Enterprise systems although their detections were performed on SCA systems [66].

We tend to improve the accuracy of COLAR tool for RESTAPI linguistic antipatterns correction. An interesting research area is to investigate the number of changes adopted by each REST API service and further investigate that which API are

survived for a longer period. Another, investigation is the collection of errors reported for a specific REST API that will help to correlate the errors with the change reported in revision history of REST API providers. There could be the improvement in the correction algorithm dynamically after linking correction definitions with user perception as input and design REST APIs as per developer's perception.

There is also a need to mine the trace history of request, repose and body of the RESTful API and investigates the correlation between practical implementation and documentation available over the internet. This will help to investigate the changes each REST API providers implemented as per the IETF standards [205]. There is also a need to report the standardize databases for status codes and description after tracking past 5 years' data of REST API providers.

We are interested to investigate the effect of different NLP algorithms for checking the semantic and syntactic similarity between RESTful API. We also want to investigate the effect of linguistic antipatterns on client APIs by conducting a case study. Another good approach is to investigate the errors against each RESTful API methods reported on developer's forum to investigate the antipattern effect with respect to user response.

Chapter 8 References

[1] Object-Oriented Software Construction vol. 2: Prentice Hall New York, 1988.

[2] Papazoglou, M. P., "Service-Oriented Computing: Concepts, Characteristics and Directions", Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE), 2003.

[3] Heffner, R., Fulton, L., Gilpin, M., Peyret, H., Vollmer, K. and Stone, J,. Topic overview: Service-oriented architecture. Forrester, June, 8, p.4,2007

[4] Alonso, G., Casati, F., Kuno, H., & Machiraju, V. Web services. In Web Services (pp. 123-149). Springer, Berlin, Heidelberg,2004.

[5] Chappell, D. (2007). Introducing sca. Avalaible at http://www. davidchappell. com/articles/Introducing_SCA. pdf.

[6] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," ACM Transactions on Internet Technology (TOIT), vol. 2 , no. 2, pp. 115–150, 2002.

[7] Heß, A., Johnston, E., Kushmerick, N.:" ASSAM: A Tool for Semi-Automatically Annotating Semantic Web Services". In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 320–334. Springer, Heidelberg (2004).

[8] Fowler, M. (1999). Refactoring: improving the design of existing code. Pearson Education India.

[9] Lehnert, S. (2011). A Review of Software Change Impact Analysis.

[10] Westland, J. C. (2004). The cost behavior of software defects. Decision Support Systems, 37(2), 229-238.

[11] Kilgarriff, A. (2000). Wordnet: An electronic lexical database.

[12] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving Multi-Objective Code-Smells Correction Using Development History" Journal of Systems and Software, vol. 105, pp. 18-39, 2015.

[13]Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., & McClosky, D. "The Stanford CoreNLP natural language processing toolkit ". In Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations,2014,pp. 55-60.

[14] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics-Based Refactoring" in Software Maintenance and Reengineering, 2001. Fifth European Conference on, 2001, pp. 30-38.

[15] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic Antipatterns: What They Are and How Developers Perceive Them" Empirical Software Engineering, vol. 21, pp. 104-158, 2016.

[17] 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 Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on, 2013, pp. 268-278.

[18] Papazoglou, M. P., Traverso, P., Dustdar, S. and Leymann, F., "ServiceOriented Computing Research Roadmap", 2006.

[19] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 2nd ed. Addison-Wesley, 2003.

[20] Brown, A., Johnston, S., & Kelly, K. (2002). Using service-oriented architecture and component-based development to build web service applications. Rational Software Corporation, 6, 1-16.

[21] R., & Magazinius, A. (2010, July). Validity Threats in Empirical Software Engineering Research-An Initial Survey. In SEKE (pp. 374-379).

[22] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing Approaches to Analyse Refactoring Activity on Software Repositories" Journal of Systems and Software, vol. 86, pp. 1006-1022, 2013.

[23] F. Khomh, M. Di Penta, and D. Ptidej Team, "An Exploratory Study of the Impact of Software Changeability" 2009.

[24] Moha, N. 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, pp. 20-36, 2010.

[25] M. Salehie, S. Li, and L. Tahvildari, "A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws" in Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on, 2006, pp. 159-168.

[26] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur, "SMURF: A SVM-Based Incremental Antipattern Detection Approach" in Reverse engineering (WCRE), 2012 19th working conference on, 2012, pp. 466-475.

[27] 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, pp. 559-572, 2011.

[28] R. Wangberg, "A Literature Review on Code Smells and Refactoring" 2010.

[29] 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, pp. 179-202, 2011.

[30] G. Rasool and Z. Arshad, "A Review of Code Smell Mining Techniques" Journal of Software: Evolution and Process, vol. 27, pp. 867-895, 2015.

[31] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach" Science of Computer Programming, vol. 74, pp. 470-495, 2009.

[32] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone Evolution: A Systematic Review" Journal of software: Evolution and Process, vol. 25, pp. 261-283, 2013.

[33] D. Rattan, R. Bhatia, and M. Singh, "Software Clone Detection: A Systematic Review" Information and Software Technology, vol. 55, pp. 1165-1199, 2013.

[34] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research" Queen's School of Computing TR, vol. 541, pp. 64-68, 2007.

[35] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering" IEEE Transactions on Software Engineering, vol. 38, pp. 1276-1304, 2012.

[36] W. C. Wake, Refactoring Workbook: Addison-Wesley Professional, 2004.

[37] M. Mantyla, J. Vanhanen, and C. Lassenius, "A Taxonomy and An Initial Empirical Study of Bad Smells in Code" in Software Maintenance. ICSM 2003. Proceedings. International Conference on, 2003, pp. 381-384.

[38] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, "Mining the Relationship Between Antipatterns Dependencies and Fault-Proneness" in Reverse Engineering (WCRE), 2013 20th Working Conference on, 2013, pp. 351-360.

[39] J. Bansiya and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment" IEEE Transactions on software engineering, vol. 28, pp. 4-17, 2002.

[40] 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.

[41] M. A. Laguna and Y. Crespo, "A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring" Science of Computer Programming, vol. 78, pp. 1010-1034, 2013.

[42] T. Mens and T. Tourwé, "A Survey of Software Refactoring" IEEE Transactions on software engineering, vol. 30, pp. 126-139, 2004.

[43] R. Wangberg, "A Literature Review on Code Smells and Refactoring" 2010.

[44] M. Misbhauddin and M. Alshayeb, "UML Model Refactoring: A Systematic Literature Review" Empirical Software Engineering, vol. 20, pp. 206-251, 2015.

[45] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from Applying the Systematic Literature Review Process Within the Software Engineering Domain" Journal of systems and software, vol. 80, pp. 571-583, 2007.

[46] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic Literature Reviews in Software Engineering–A Systematic Literature Review" Information and software technology, vol. 51, pp. 7-15, 2009.

[47] F. Palma, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, "Specification and Detection of SOA Antipatterns In Web Services" in European Conference on Software Architecture 2014, pp. 58-73.

[48] C. Mateos, J. M. Rodriguez, and A. Zunino, "A Tool to Improve Code- First Web Services Discoverability Through Text Mining Techniques" Software: Practice and Experience, vol. 45, pp. 925-948, 2015.

[49] J. L. Ordiales Coscia, C. Mateos, M. Crasso, and A. Zunino, "Antipattern Free Code-First Web Services for State-Of-The-Art Java WSDL Generation Tools" International Journal of Web and Grid Services, vol. 9, pp. 107-126, 2013.

[50] G. Salvatierra, C. Mateos, M. Crasso, and A. Zunino, "Towards A Computer-Assisted Approach for Migrating Legacy Systems To SOA" in International Conference on Computational Science and Its Applications, 2012, pp. 484-497.

[51] S. Keele, "Guidelines for Performing Systematic Literature Reviews in Software Engineering" in Technical report, Ver. 2.3 EBSE Technical Report. EBSE, ed: sn, 2007.

[52] A. Ouni, R. Gaikovina Kula, M. Kessentini, and K. Inoue, "Web Service Antipatterns Detection Using Genetic Programming" in Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, 2015, pp. 1351-1358.

[53] J. L. O. Coscia, C. Mateos, M. Crasso, and A. Zunino, "Avoiding WSDL Bad Practices in Code-First Web Services" in Proceedings of the 12th Argentine Symposium on Software Engineering (ASSE2011)-40th JAIIO, 2011, pp. 1-12.

[54] Ouni, A., Kessentini, M., Inoue, K., & Cinnéide, M. O. "Search-based web service antipatterns detection". IEEE Transactions on Services Computing, 10(4), 603-617,2017.

[55] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, and A. Tiberghien, "From A Domain Analysis to The Specification and Detection of Code and Design Smells" Formal Aspects of Computing, vol. 22, pp. 345-361, 2010.

[56] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc, "Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach" in International Conference on Service-Oriented Computing, 2014, pp. 230-244.

[57] F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, and G. Tremblay, "Are Restful APIs Well-Designed? Detection of Their Linguistic (Anti) Patterns" in International Conference on Service-Oriented Computing, 2015, pp. 171-187.

[58] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting Bugs Using Antipatterns" in Software Maintenance (ICSM), 2013 29th IEEE International Conference on, 2013, pp. 270-279.

[59] L. Guerrouj, Z. Kermansaravi, V. Arnaoudova, B. C. Fung, F. Khomh, G. Antoniol, et al., "Investigating the Relation Between Lexical Smells and Change- and Fault-Proneness: An Empirical Study" Software Quality Journal, pp. 1-30, 2015.

[60] F. A. Fontana, V. Ferme, and S. Spinelli, "Investigating the Impact of Code Smells Debt on Quality Code Evaluation" in Managing Technical Debt (MTD), 2012 Third International Workshop on, 2012, pp. 15-22.

[61] J. Al Dallal, "Identifying Refactoring Opportunities in Object-Oriented Code: A Systematic Literature Review" Information and software Technology, vol. 58, pp. 231-249, 2015.

[62] R. Marinescu, "Measurement and Quality in Object-Oriented Design" in Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, 2005, pp. 701-704.

[63] Z. Ujhelyi, G. Szőke, Á. Horváth, N. I. Csiszár, L. Vidács, D. Varró, et al., "Performance Comparison of Query-Based Techniques for Antipattern Detection" Information and Software Technology, vol. 65, pp. 147-165, 2015.

[64] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Gueheneuc, "A New Family of Software Antipatterns: Linguistic Antipatterns" in Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, 2013, pp. 187-196.

[65] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws" in Software Maintenance Proceedings. 20th IEEE International Conference on, 2004, pp. 350-359.

[66] F. Palma, M. Nayrolles, N. Moha, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "SOA Antipatterns: An Approach for Their Specification and Detection" International Journal of Cooperative Information Systems, vol. 22, p. 1341004, 2013.

[67] Nayrolles, Mathieu, Naouel Moha, and Petko Valtchev. "Improving SOA antipatterns detection in Service Based Systems by mining execution traces." In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 321-330. IEEE, 2013.

[68] F. Palma, L. An, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Investigating the Change-Proneness of Service Patterns and Antipatterns" in Service-Oriented

Computing and Applications (SOCA), 2014 IEEE 7th International Conference on, 2014, pp. 1-8.

[69] P. Danphitsanuphan and T. Suwantada, "Code Smell Detecting Tool and Code Smell-Structure Bug Relationship" in Engineering and Technology (S-CET), 2012 Spring Congress on, 2012, pp. 1-5.

[70] 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, pp. 841-861, 2014.

[71] A. Stoianov and I. Şora, "Detecting Patterns and Antipatterns In Software Using Prolog Rules" in Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on, 2010, pp. 253-258.

[72] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting Software Modularity Violations" in Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 411-420.

[73] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Detection of Embedded Code Smells in Dynamic Web Applications" in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 282-285.

[74] A. Yamashita and L. Moonen, "Exploring the Impact of Inter-Smell Relations on Software Maintainability: An Empirical Study" in Software Engineering (ICSE), 2013 35th International Conference on, 2013, pp. 682-691.

[75] R. Marinescu and D. Ratiu, "Quantifying the Quality of Object-Oriented Design: The Factor Strategy Model" in Reverse Engineering, 2004. Proceedings. 11th Working Conference on, 2004, pp. 192-201.

[76] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical Signatures of Antipatterns: An Approach Based On B-Splines" in Software maintenance and reengineering (CSMR), 2010 14th European Conference on, 2010, pp. 248-251.

[77] A. Yamashita and L. Moonen, "To What Extent Can Maintenance Problems Be Predicted by Code Smell Detection? – An Empirical Study" Information and Software Technology, vol. 55, pp. 2223-2242, 2013.

[78] C. Mateos, M. Crasso, A. Zunino, and J. L. O. Coscia, "Revising WSDL Documents: Why and How, Part 2" IEEE Internet Computing, vol. 17, pp. 46-53, 2013.

[79] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability Defects Detection and Correction: A Multi-Objective Approach" Automated Software Engineering, pp. 1-33, 2013.

[80] 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 Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, 2013, pp. 167-176.

[81] R. Marinescu and C. Marinescu, "Are the Clients of Flawed Classes (Also) Defect-prone?" in Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on, 2011, pp. 65-74.

[82] M. Petticrew and H. Roberts, Systematic Reviews in The Social Sciences: A Practical Guide: John Wiley & Sons, 2008.

[83] J. Kreimer, "Adaptive Detection of Design Flaws" Electronic Notes in Theoretical Computer Science, vol. 141, pp. 117-136, 2005.

[84] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Gueheneuc, "Can Lexicon Bad Smells Improve Fault Prediction?" in Reverse Engineering (WCRE), 2012 19th Working Conference on, 2012, pp. 235-244.

[85] A. Bandi, B. J. Williams, and E. B. Allen, "Empirical Evidence of Code Decay: A Systematic Mapping Study" in Reverse Engineering (WCRE), 2013 20th Working Conference on, 2013, pp. 341-350.

[86] H. Liu, X. Guo, and W. Shao, "Monitor-Based Instant Software Refactoring" IEEE Transactions on Software Engineering, vol. 39, pp. 1112-1126, 2013.

[87] A. Trifu, O. Seng, and T. Genssler, "Automated Design Flaw Correction in Object-Oriented Systems" in Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on, 2004, pp. 174-183.

[88] F. Jaafar, F. Khomh, Y.-G. Guéhéneuc, and M. Zulkernine, "Antipattern Mutations and Fault-Proneness" in Quality Software (QSIC),14th International Conference on, 2014, pp. 246-255.

[89] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, and S. Hamel, "IDS: An Immune-Inspired Approach for The Detection of Software Design Smells" in Quality of Information and Communications Technology (QUATIC),Seventh International Conference on the, 2010, pp. 343-348.

[90] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What You Like in Design Use to Correct Bad-Smells" Software Quality Journal, vol. 21, pp. 551-571, 2013.

[91] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, "Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains" in Software Maintenance (ICSM),29th IEEE International Conference on, 2013, pp. 260-269.

[92] I. Polášek, P. Líška, J. Kelemen, and J. Lang, "On Extended Similarity Scoring and Bit-Vector Algorithms for Design Smell Detection" in Intelligent Engineering Systems (INES), IEEE 16th International Conference on, 2012, pp. 115-120.

[93] N. Maneerat and P. Muenchaisri, "Bad-Smell Prediction from Software Design Model Using Machine Learning Techniques" in Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on, 2011, pp. 331-336.

[94] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort" IEEE Transactions on Software Engineering, vol. 38, pp. 220-235, 2012.

[95] Y. Luo, A. Hoss, and D. L. Carver, "An Ontological Identification of Relationships Between Antipatterns and Code Smells" in Aerospace Conference, 2010 IEEE, 2010, pp. 1-10.

[96] M. V. Mantyla, J. Vanhanen, and C. Lassenius, "Bad Smells-Humans as Code Critics. Proceedings. 20th IEEE International Conference on Software Maintenance, 2004, pp. 399-408.

[97] G. Ganea, I. Verebi, and R. Marinescu, "Continuous Quality Assessment With inCode" Science of Computer Programming, 2015.

[98] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic Detection of Bad Smells in Code: An Experimental Assessment" Journal of Object Technology, vol. 11, pp. 5:1-38, 2012.

[99] B. F. dos Santos Neto, M. Ribeiro, V. T. da Silva, C. Braga, C. J. P. de Lucena, and E. de Barros Costa, "AutoRefactoring: A Platform to Build Refactoring Agents" Expert Systems with Applications, vol. 42, pp. 1652-1664, 2015.

[100] J. L. O. Coscia, C. Mateos, M. Crasso, and A. Zunino, "Refactoring Code-First Web Services for Early Avoiding WSDL Antipatterns: Approach and comprehensive assessment" Science of Computer Programming, vol. 89, pp. 374-407, 2014.

[101] M. T. Llano and R. Pooley, "UML Specification and Correction of Object-Oriented Antipatterns" ICSEA'09. Fourth International Conference on Software Engineering Advances, 2009, pp. 39-44.

[102] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-Based Refactoring: Towards Semantics Preservation" in Software Maintenance (ICSM), 2012 28th IEEE International Conference on, 2012, pp. 347-356.

[103] Palomba, Fabio, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. "Mining version histories for detecting code smells." *IEEE Transactions on Software Engineering* 41, no. 5 (2015): 462-489..

[104] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An Experimental Investigation on The Innate Relationship Between Quality and Refactoring" Journal of Systems and Software, vol. 107, pp. 1-14, 2015.

[105] B. Walter and T. Alkhaeir, "The Relationship Between Design Patterns and Code Smells: An Exploratory Study" Information and Software Technology, vol. 74, pp. 127-142, 2016.

[106] J. M. Rodriguez, M. Crasso, C. Mateos, and A. Zunino, "Best Practices for Describing, Consuming, And Discovering Web Services: A Comprehensive Toolset" Software: Practice and Experience, vol. 43, pp. 613-639, 2013.

[107] H. S. de Andrade, E. Almeida, and I. Crnkovic, "Architectural Bad Smells in Software Product Lines: An Exploratory Study" in Proceedings of the WICSA 2014 Companion Volume, 2014, p. 12.

[108] 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, pp. 1143-1191, 2016.

[109] I. Bashir and A. L. Goel, Testing Object-Oriented Software: Life Cycle Solutions: Springer Science & Business Media, 2012.

[110] C. Wohlin, "Guidelines for Snowballing in Systematic Literature Studies and A Replication in Software Engineering" in Proceedings of the 18th international conference on evaluation and assessment in software engineering, 2014, p. 38.

[111] D. S. Cruzes and T. Dybå, "Research Synthesis in Software Engineering: A Tertiary Study" Information and Software Technology, vol. 53, pp. 440-455, 2011.

[112] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the Effect of Code Smells on Maintenance Effort" IEEE Transactions on Software Engineering, vol. 39, pp. 1144-1156, 2013.

[113] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An Exploratory Study of The Impact of Antipatterns On Class Change-And Fault-Proneness" Empirical Software Engineering, vol. 17, pp. 243-275, 2012.

[114] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some Code Smells Have A Significant But Small Effect On Faults" ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 23, p. 33, 2014.

[115] 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, pp. 544-558, 2016.

[116] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, "Design Guidelines For Domain Specific Languages" arXiv preprint arXiv:1409.2378, 2014.

[117] L. H. Etzkorn, S. E. Gholston, J. L. Fortune, C. E. Stein, D. Utley, P. A. Farrington, et al., "A Comparison of Cohesion Metrics for Object-Oriented Systems" Information and Software Technology, vol. 46, pp. 677-687, 2004.

[118] T. Erl, Service-Oriented Architecture: Concepts, Technology, And Design: Pearson Education India, 2005.

[119] Jaafar, Fehmi, Angela Lozano, Yann-Gaël Guéhéneuc, and Kim Mens. "Analyzing software evolution and quality by extracting Asynchrony change patterns." Journal of Systems and Software 131 ,2017pp. 311-322.

[120] Petrillo, F., Merle, P., Moha, N., & Guéhéneuc, Y.-G. "Are REST APIs for cloud computing well-designed? An exploratory study". In International Conference on Service-Oriented Computing, pp. 157-170. Springer, Cham, 2016.

[121] Wang, Hanzhang, Marouane Kessentini, and Ali Ouni. "Prediction of Web Services Evolution." In International Conference on Service-Oriented Computing, pp. 282-297. Springer, Cham, 2016.

[122] Sjoberg, Dag IK, Tore Dyba, and Magne Jorgensen. "The future of empirical methods in software engineering research." In Future of Software Engineering, 2007. FOSE'07, pp. 358-378. IEEE, 2007.

[123] Breivold, H. P., & Larsson, M. Component-based and service-oriented software engineering: Key concepts and principles. In Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on (pp. 13-20). IEEE.

[124] Baghdadi, Y. Service-oriented software engineering: a guidance framework for service engineering methods. International Journal of Systems and Service-Oriented Engineering (IJSSOE), 2015, 5(2), 1-19.

[125] Papazoglou, Mike P. "Service-oriented computing: Concepts, characteristics and directions." In Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on, pp. 3-12. IEEE, 2003.

[126] Deb, Kalyanmoy, et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II." IEEE transactions on evolutionary computation 6.2 (2002): 182-197.

[128] G. Booch, Object-Oriented Analysis & Design with Application: Pearson Education, 2006.

[129] M. Fowler and K. Beck, Refactoring: Improving the Design of Existing Code: Addison-Wesley Professional, 1999.

[130] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, et al., "When and Why Your Code Starts to Smell Bad" in Proceedings of the 37th International Conference on Software Engineering-Volume 1, 2015, pp. 403-414.

[131] Lehnert, S. (2011, September). A taxonomy for software change impact analysis. In Proceedings of the 12th International Workshop on Principles of

Software Evolution and the 7th annual ERCIM Workshop on Software Evolution(pp. 41-50). ACM.

[132] Kr´al, J., Zemliˇ ˇ cka, M.: Crucial Service-Oriented Antipatterns, vol. 2, pp. 160–171. International Academy, Research and Industry Association, IARIA (2008)

[134] C. Izurieta, Decay and Grime Buildup in Evolving Object-Oriented Design Patterns: Colorado State University, 2009.

[136] C. Gravino, M. Risi, G. Scanniello, and G. Tortora, "Does the Documentation of Design Pattern Instances Impact On Source Code Comprehension? Results From Two Controlled Experiments"18th Working Conference on Reverse Engineering (WCRE), 2011, pp. 67-76.

[137] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy, "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance" IEEE Transactions on Software Engineering, vol. 28, pp. 595-606, 2002.

[138] S. Biffl and W. Gutjahr, "Influence of Team Size and Defect Detection Technique on Inspection Effectiveness" in Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International, 2001, pp. 63-75.

[139]Monteiro, M. P., & Fernandes, J. M. Refactoring a Java code base to AspectJ: An illustrative example. In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05),2005, (pp. 1063-6773/05). IEEE.

[142] El Boussaidi, Ghizlane, Duc-Loc Huynh, and Naouel Moha. "Detection of Design Defects: Formal Concept Analysis and Metrics." (2005).

[143] M. J. Munro, "Product Metrics for Automatic Identification Of" Bad Smell" Design Problems In Java Source-Code" in Software Metrics, 2005. 11th IEEE International Symposium, 2005.

[144] Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave Macmillan, 2005.

[145] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. European Conference on Object Oriented Programming, 1993.

[146] Riel, A. J. Object-oriented heuristics,1993 (Vol. 335). Reading: AddisonWesley.

[147] Abbes, M., Khomh, F., Gueheneuc, Y. G., & Antoniol, G. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In Software maintenance and reengineering (CSMR), 2011 15th European conference on (pp. 181-190).

[148] Mäntylä, Mika V., and Casper Lassenius. "Subjective evaluation of software evolvability using code smells: An empirical study." Empirical Software Engineering 11, no. 3: 395-431,2006.

[149] Arcelli, Davide, Vittorio Cortellessa, and Catia Trubiani. "Antipattern-based model refactoring for software performance improvement." In Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, pp. 33-42. ACM, 2012.

[150]Kaur, H., & Kaur, P. J.. A Study on Detection of Antipatterns in Object-Oriented Systems. International Journal of Computer Applications,93(5),2014.

[151] Erlikh, L. (2000). "Leveraging legacy system dollars for E-business". (IEEE) IT Pro, May/June 2000, 17-23.

[152] Moha, N., Gueheneuc, Y. G., & Leduc, P. Automatic generation of detection algorithms for design defects. In 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06) (pp. 297-300),2006.

[153] Rodriguez, J. M., Crasso, M., Zunino, A., & Campo, M. Improving Web Service descriptions for effective service discovery. Science of Computer Programming, 75(11), 1001-1021,2010.

[154]M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau, "An empirical study on web service evolution," in Web Services (ICWS), IEEE International Conference on. IEEE, pp. 49–56,2011.

[155] M. M. Lehman and L. A. Belady, Program evolution: processes of software change. Academic Press Professional, Inc., 1985.

[156] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring," Journal of Software: Evolution and Process, vol. 18, no. 2, pp. 83–107, 2006.

[157]M. Laitinen, “Framework maintenance: Vendor viewpoint,” Object-Oriented Application Frameworks: Problems and Perspectives, ME Fayad, DC Schmidt, RE Johnson (eds), Wiley & Sons, 1999.

[158]R. Lammel,¨ E. Pek, and J. Starek, “Large-scale, ast-based api-usage analysis of open-source java projects,” in Proceedings of the 2011 ACM Symposium on Applied Computing. ACM, pp. 1317–1324,2011.

[159]S. Blank, “Api integration pain survey results,” 2011.

[160]J. Webber, S. Parastatidis, and I. Robinson, REST in practice: Hyper-media and systems architecture. ” O’Reilly Media, Inc.”, 2010.

[161]A. Demange, N. Moha, and G. Tremblay, “Detection of SOA patterns,” in International Conference on Service-Oriented Computing. Springer, 2013, pp. 114–130.

[162]T. Erl and S. D. Patterns, “Prentice hall ptr,” Upper Saddle River, NJ, p. 65, 2009.

[163]T. Erl, B. Carlyle, C. Pautasso, and R. Balasubramanian, SOA with REST: Principles, Patterns &Constraints for Building Enterprise Solutions with REST. Prentice Hall Press, 2012.

[164] F. Petrillo, P. Merle, N. Moha, and Y.-G. Gueh´eneuc,´ “Towards a rest cloud computing lexicon,” in 7th International Conference on Cloud Computing and Services Science, CLOSER , 2017.

[165][Zhou, W., Li, L., Luo, M., & Chou, W. (2014, May). REST API design patterns for SDN northbound API. In Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on (pp. 358-365). IEEE.

[166] R. Jabangwe et al., “An exploratory study of software evolution and quality: Before, during and after a transfer,” in Global Software Engineering (ICGSE), 2012 IEEE Seventh International Conference on. IEEE, 2012, pp. 41–50.

[167] 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 (and whether the smells go away),” IEEE Transactions on Software Engineering, 2017.

[168] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, “Tracking the software quality of android applications along their evolution (t),” in Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. IEEE, 2015, pp. 236–247.

[169] L. Li and W. Chou, “Design and describe rest api without violating rest: A petri net based approach,” in Web Services (ICWS), IEEE International Conference on. IEEE, 2011, pp. 508–515.

[170] C. Rodr´ıguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, “Rest apis: a large-scale analysis of compliance with principles and best practices,” in International Conference on Web Engineering. Springer, 2016, pp. 21–39.

[171] F. Haupt, F. Leymann, A. Scherer, and K. Vukojevic-Haupt, “A frame-work for the structural analysis of rest apis,” in Software Architecture (ICSA),IEEE International Conference on. IEEE, 2017, pp. 55– 58.

[172] M. Athanasopoulos and K. Kontogiannis, “Extracting rest resource models from procedure-oriented service interfaces,” Journal of Systems and Software, vol. 100, pp. 149–166, 2015.

[173] K. Mohamed and D. Wijesekera, “Performance analysis of web services on mobile devices,” Procedia Computer Science, vol. 10, pp. 744–751, 2012.

[174] F. Simon, F. Steinbruckner, and C. Lewerentz, “Metrics based refactor-ing,” in Software Maintenance and Reengineering, Fifth European Conference on. IEEE, 2001, pp. 30–38.

[175] X. Ge, Q. L. DuBose, and E. Murphy-Hill, “Reconciling manual and automatic refactoring,” in Software Engineering (ICSE), 2012 34th International Conference on. IEEE, 2012, pp. 211–221.

[176] I. Ivkovic and K. Kontogiannis, “A framework for software architecture refactoring using model transformations and semantic annotations,” in Software Maintenance and Reengineering, 2006. CSMR 2006. Proceed-ings of the 10th European Conference on. IEEE, 2006, pp. 10–pp.

[177] N. Moha, “Detection and correction of design defects in object-oriented designs,” in Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion. ACM, 2007, pp. 949–950.

[178] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, “Design defects detection and correction by example,” in Program Comprehension (ICPC), 2011 IEEE 19th International Conference on. IEEE, 2011, pp. 81–90.

[179] N. Moha, A. Rouane Hacene, P. Valtchev, and Y.-G. Gueh´eneuc,´ “Refactorings of design defects using relational concept analysis,” Formal Concept Analysis, pp. 289–304, 2008.

[180] N. Moha, J. Rezgui, Y.-G. Gueh´eneuc,´ P. Valtchev, and G. El Boussaidi, “Using fca to suggest refactorings to correct design defects,” in Concept Lattices and Their Applications. Springer, 2008, pp. 269–275.

[181]G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, “Methodbook: Recommending move method refactorings via relational topic models,” IEEE Transactions on Software Engineering, vol. 40, no. 7, pp. 671–694, 2014.

[182]B. F. dos Santos Neto, M. Ribeiro, V. T. da Silva, C. Braga, C. J. P. de Lucena, and E. de Barros Costa, “Autorefactoring: A platform to build refactoring agents,” Expert Systems with Applications, vol. 42, no. 3, pp. 1652–1664, 2015.

[183] N. Tsantalis and A. Chatzigeorgiou, “Ranking refactoring suggestions based on historical volatility,” in Software Maintenance and Reengi-neering (CSMR), 2011 15th European Conference on. IEEE, 2011, pp. 25–34.

[184] T. Mens, G. Taentzer, and O. Runge, “Analyzing refactoring dependencies using graph transformation,” Software and Systems Modeling, vol. 6, no. 3, p. 269, 2007.

[185] C. Pautasso, “Some rest design patterns (and antipatterns),” 2009.

[186] J. Purushothaman, RESTful Java Web Services. Packt Publishing Ltd, 2015.

[187] J. Sandoval, Restful java web services: Master core rest concepts and create restful web services in java. Packt Publishing Ltd, 2009.

[188] S. Allamaraju, Restful web services cookbook: solutions for improving scalability and simplicity. ” O’Reilly Media, Inc.”, 2010.

[189] L. Richardson and S. Ruby, RESTful web services.” O’Reilly Media, Inc.”, 2008.

[190] T. Espinha, A. Zaidman, and H.-G. Gross, “Web api growing pains: Loosely coupled yet strongly tied,” Journal of Systems and Software, vol. 100, pp. 27–43, 2015.

[191] R. Daigneau, Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services. Addison-Wesley, 2011.

[192]G. Salvatierra, C. Mateos, M. Crasso, and A. Zunino, “Towards a computer assisted approach for migrating legacy systems to soa,” Computational Science and Its Applications–ICCSA 2012, pp. 484–497, 2012.

[193] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, “Discoverability antipatterns: frequent ways of making undiscoverable web service descriptions,” in Proceedings of the 10th Argentine Symposium on Software Engineering (ASSE2009)-38th JAIIO, 2009, pp. 1–15.

[194] M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo, “Revising wsdl documents: Why and how,” IEEE Internet Computing, vol. 14, no. 5,48–56, 2010.

[196] S. Tilkov. Rest antipatterns. [Online]. Available: https://www.infoq.com/articles/rest-antipatterns

[197] F. Jaafar, Y.-G. Gueh´eneuc,´ S. Hamel, F. Khomh, and M. Zulkernine, “Evaluating the impact of design pattern and antipattern dependencies on changes and faults,” Empirical Software Engineering, vol. 21, no. 3,896–931, 2016.

[198] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, “An exploratory study of the impact of code smells on software change-proneness,” in Reverse Engineering, 2009. WCRE’09. 16th Working Conference on. IEEE, 2009, pp. 75–84.

[199] B. Costa, P. F. Pires, F. C. Delicato, and P. Merson, “Evaluating rest architectures—approach, tooling and guidelines,” Journal of Systems and Software, vol. 112, pp. 156–180, 2016.

[200] Spinellis, Diomidis. "Version control systems." *IEEE Software*22, no. 5 (2005): 108-109.

[201] S. Otte, "Version control systems," Computer Systems and Telematics, Institute of Computer Science, Freie Universitat,¨ Berlin, Germany, 2009.

[202] Kuttal, S. K., Sarma, A., & Rothermel, G. (2014). "On the benefits of providing versioning support for end users: an empirical study". ACM Transactions on Computer-Human Interaction (TOCHI), 21(2), 9.

[203] Herzig, K., & Zeller, A. "The impact of tangled code changes". In Proceedings of the 10th Working Conference on Mining Software Repositories (pp. 121-130),2013 IEEE Press.

[204] B. Walter and T. Alkhaeir, "The relationship between design patterns and code smells: An exploratory study," Information and Software Technology, vol. 74, pp. 127–142, 2016.

[205] F. Reschke. Hyper text transfer protocol. [Online]. Available: https://tools.ietf.org/html/rfc7230#section-6.1

[206] Mass_e, M.: REST API Design Rulebook. O'Reilly (2012)

[207] Berners-Lee, T., Fielding, R.T., Masinter, L.: Uniform Resource Identi_er (URI):Generic Syntax ,2005

[208] Fredrich, T.: RESTful Service Best Practices: Recommendations for Creating WebServices ,http://www.restapitutorial.com/resources.html,May 2012.

[208] Maleshkova, M., Pedrinaci, C.,& Domingue, J."Investigating web apis on the world wide web". In Web Services (ECOWS), IEEE 8th European Conference on (pp. 107-114). IEEE,2010.

[209] Neamtiu, I., Foster, J. S., & Hicks, M. "Understanding source code evolution using abstract syntax tree matching". ACM SIGSOFT Software Engineering Notes, 30(4), 1-5,2005.

[210] Rodriguez, J. M., Crasso, M., Zunino, A., & Campo, M. "Discoverability antipatterns: frequent ways of making undiscoverable Web Service descriptions". In Proc. 10th Argentine Symp. Software Eng (pp. 1-15),2009.

[211]Fluri, B., Wuersch, M., PInzger, M., & Gall, H.” Change distilling: Tree differencing for fine-grained source code change extraction”. IEEE Transactions on software engineering, 33(11).2007.

[212] Ouni, A., Daagi, M., Kessentini, M., Bouktif, S., & Gammoudi, M. M. “A Machine Learning-Based Approach to Detect Web Service Design Defects”.IEEE International Conference on Web Services (ICWS), (pp. 532-539). IEEE.

[213]De Lucia, A., Di Penta, M., & Oliveto, R. (2011). Improving source code lexicon via traceability and information retrieval. IEEE Transactions on Software Engineering, 37(2), 205-227.

[214] Abebe, S., and P. Tonella, Towards the extraction of domain concepts from the identi_ers, in Proceedings of the Working Conference on Reverse Engineering (WCRE), pp. 77{86, 2011.

[215] Abebe, S. L., Alicante, A., Corazza, A., & Tonella, P. (2013). Supporting concept location through identifier parsing and ontology extraction. Journal of Systems and Software, 86(11), 2919-2938.

[216] De Lucia, A., M. Di Penta, and R. Oliveto, Improving source code lexicon via traceability and information retrieval, IEEE Transactions on Software Engineering, 37 (2), 205{227, 2011.

[217] Wampler, D. (2011). Scala web frameworks: Looking beyond lift. IEEE Internet Computing, 15(5), 87-94.

[218] Shull, F., Singer, J., & Sjøberg, D. I. (Eds.). (2007). Guide to advanced empirical software engineering. Springer Science & Business Media.

[219] Groves, R. M., Fowler Jr, F. J., Couper, M. P., Lepkowski, J. M., Singer, E., & Tourangeau, R. (2011). Survey methodology(Vol. 561). John Wiley & Sons.

[220] De Lucia, A., Oliveto, R., Zurolo, F., & Di Penta, M. (2006, June). Improving comprehensibility of source code via traceability information: a controlled experiment. In Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on (pp. 317-326). IEEE.

[221] Deissenboeck, Florian, and Markus Pizka. "Concise and consistent naming." Software Quality Journal 14, no. 3 (2006): 261-282.

[222] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering traceability links between code and documentation. IEEE transactions on software engineering, 28(10), 970-983.

[223] Marcus, A., & Maletic, J. I. (2003, May). Recovering documentation-to-source-code traceability links using latent semantic indexing. In Software Engineering, 2003. Proceedings. 25th International Conference on (pp. 125-135). IEEE.

[224] Riebisch, M. (2004, May). Supporting evolutionary development by feature models and traceability links. In Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the (pp. 370-377). IEEE.

[225] Godbole, N., Srinivasaiah, M., & Skiena, S. Large-Scale Sentiment Analysis for News and Blogs. Icwsm, 7(21), 219-222,2007.

[226] Weissgerber, P., & Diehl, S. Identifying refactorings from source-code changes. In Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on (pp. 231-240). IEEE.

[227] Kim, M., Gee, M., Loh, A., & Rachatasumrit, N. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (pp. 371-372). ACM.

[228] R. Masson. How REST replaced SOAP on the Web: What it means to you,. [Online]. Available: https://www.infoq.com/articles/rest-soap.

[229] Malpohl, Guido, James J. Hunt, and Walter F. Tichy. Renamingdetection. *Automated Software Engineering* 10, no. 2 (2003): 183-202.

Appendices

Appendix I: Primary studies used for SLR

Title of the Paper	Reference Number	Conference	Journal Name
Specification and detection of SOA Antipattern in Web Services	47	ECSA2014, Book Chapter in Springer	not Applicable
A tool to improve Code first Web service Discoverability through text miing techniques	48	Not applicab	Softwre practise and experience
Mining the Relationship between Antipatterns Dependencies and Fault-Proneness	38	WCRE	not Applicable
Towards a Computer Assisted approach for migrating legacy system	50	ICCSA 2012	International Journal of web and Grid Services
detecting WSDl bad practise in code first web service	51	Not applicab	International Journal of web and Grid Services
Web service Antipattern detection using genetic programming	52	Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation	not Applicable
Avoiding WSDl bad Practise in Code first web service	53	ASSE	not Applicable
Search Based Web Service Antipattern detection	54	Not applicab	IEEE Transaction Service Coing mpu
From a domain analysis to the specification and detection of code and design smell	55	Not applicab	Formal Aspect of Computing
Detection of REST patterns and AP :A heuristic based approach	56	ICSOC	not Applicable
Are RESTful APIs well designed?Detection of their linguistic AP	57	ICSOC	not Applicable
SMURF;An SVm based incremental Reverse Engineering Approach for Antipattern Detection	26	WCRE	not Applicable
Predicting Bugs using AP	58	ICSM	not Applicable
Investigating the impact of Code smell debt on Quality code evaluation	60	Third International Workshop on Managing Technical Debt	not Applicable
Identifying refactoring opportunities in OOCode: A systematic Literature Review	61	Not applicab	Journal of Information and Software Technology
A metric Based Heuritic Framework to detect Object Orineted Design Flaws	62	ICPC	not Applicable
Performance Comparison of Query Based Techniques for Antipatttern Detection	63	Not applicab	IST
A review of Code smell Mining Tecniques	30	Not applicab	Jurnal of Software evolution and Process
Detection Strategies: Metric based rules for detecting Design flaws	65	ICSM	not Applicable
Detecting Design Flaws via Metris in OOSE	16	International conference on Exhibtion on 2001	not Applicable
Linguistic AP: What they are and how developers percieve them	15	Not applicab	Empirical Software Engineering

Antipattern free code first web Servcie for state of the art Java2WSDl tool	49	Not applicab	International Journal of web and Grid Services
SOA AP: An Approach for their Specification and Detection	66	ICSOC	not Applicable
Investigating the Change pronesness of Service Patterns nd AP	68	ICSOC	not Applicable
Code smell Detecting tool And code Smell Strucutre Bug RelatioShip	69	Spring Congress on engineering and Technology	not Applicable
A Cooperative paprallel serach based software engineering Approach for Code Smell Detection	70	Not applicab	IEEE transaction on Software engineering
Décor:A method for the Specification and Detection of Code snd DS	24	Not applicab	IEEEn transaction on SWE
A Metric Based Heuristic to detect Object Oriented design flaws	25	ICPC	not Applicable
Improving SOA AP in Service Based System by Mining exceution traces	67	WCRE	not Applicable
Detecting Patterns and AP Using Prolog Rules	71	ICC-CONIT	not Applicable
detecting Software Modularity Violation	72	ICSE	not Applicable
Detection of embeded CS in Dynamic Web Application	73	ASE	not Applicable
What you like in Design rule to correct Bad Smell	90	Not applicab	Software quality journal
Metric Based rules for detecting Design Flaws	75	ICSM	not Applicable
AN EXPLORATORY STUDY OF THE IMPACT OF SOFTWARE CHANGEABILITY	23	Not applicab	Journal of System and software
BDTEX:A GQM based Bayesian Approach for the detection of AP	27	Not applicab	Journal of System and software
Numerical Signature of AP:An Approach based on B-splines	76	CSMR	not Applicable
Quantifying the effect of code smell on maintainance effort	112	Not applicab	IEEE transcation on software engineering
Revising WSDl Document : Why and How part II	78	NA	IEEE internet Computing
Detecting bad Smel in Soure code using Change History Information	17	ASE	not Applicable
Exploring the impact of Intersmell relations on Software Maintainability	74	ICSE	not Applicable
Are the client of flawed Classes also defect Prone?	81	SCAM	not Applicable
Improving the percision of Fowler defination of Bad Smell	82	Software engineering work Shop	not Applicable

Adaptive detection of Desing flaws	83	Not applicab	Electronic Notes in Theoretical Computer Science
Can Lexicon Bad smells Improves Fault Prediction	84	wcre	not Applicable
To What extent can miantiance problem be prediced by Code smell detection	77	Not applicab	not Applicable
maintainability of defect detection and correction	79	ASE	not Applicable
Automated Design flaw Correction in Object Oriented System	87	CSMR	not Applicable
Antipattern Mutation And faut proness	88	QSIC	not Applicable
IDS:An Immune Inspired Approach for the detection of DS	89	Quality Information And commuication technology	not Applicable
Empirical Evidenece of code decay:A systematic Mapping	85	WCRE	not Applicable
Monitor based Instant Software refactoring	86	Not applicab	IEEE Transaction on Software Engineering
On Extended Similarity Scoring and Bit-vector Algorithms for Design Smell Detection	92	International Conference on Intelligent Agents	not Applicable
Bad-smell Prediction from Software Design Model Using Machine Learning Techniques	93	jcse	not Applicable
Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort	94	Not applicab	IEEE Transaction on Software Engineering
Investogating the Impact of Code smell on System Quality	91	Conference on Softare Maintainance	not Applicable
An Ontological Identification of Relationships between Antipatterns and CS	95	AeroSpace Conference	not Applicable
Bad Smell Human As Code Critics	96	WCSM	not Applicable
Improving Multi Objective Code smell Correction Using development History	12	Not applicab	Journal of System and Software
A study on the relation between AP and Cost of Class Unit testing	80	CSMR	not Applicable
Continuous Quality Assesmenet with in code	97	Not applicab	not Applicable
Automatic detection of Bad smells in Code: An experimental Assesment	98	Not applicab	Journal of Object technology
Auutorefactoring :A platform to build refactoring agents	99	Not applicab	Expert System with application
Refactoring Code first Web Servie for early avoiding WSDl AP:Approach and Comprehensive assesment	100	Not applicab	Science of Computer Programming
UML Specification and Correction of Object oriented AP	101	ICSEA	not Applicable

Search Based Refactoring towards Sementic preservation	102	ICSM	not Applicable
mining version histories for detecting CS	103	Not applicab	IEEE transaction on Software engineering
An experimenta linvestigation on the innaterelationship between quality and refactoring	104	not applicab	Journal of System and software
The relation between Design pattern and CS	105		IST
Best practices for describing, consuming, and discovering web services: a comprehensive toolset	106	Not applicab	Journal of Software Practises and Evolution
A new Family of Software Antipattern:Linguistic Antipattern	64	CSMR	not Applicable
Investigating the relation between lexical smells and chage and fault proness	59		Software quality journal
Some code smells have a significant but small effect on faults	114	NA	ACM transaction
dynamic and automatic feed back threshold adaptation for code smell detetion	115	NA	IEEE transaction
An exploratory study n the impact of anitpattern on class change and fault proness	113		Empirical Software Engineering

Appendix II: Evaluation of REST API

Questionnaire for Evaluation

for the Correction of Antipatterns inREST APIs

Antipatterns may be the result of a manager or developer working under time pressure and having to solve a problem as fast as possible, they can also be the result of having applied a perfectly good pattern in a different context.

As part of the detection and correction of antipatterns in REST API, we are working on a dynamic approach for the correction of antipatterns in REST API. We develop a tool name SOCA-R (Service Oriented Correction of Antipatterns for REST APIs) that helps to correct antipatterns after the analyses of several dynamic and static properties of some REST APIs.

The purpose of this survey is to have experts evaluate correction proposed by our approach implemented in SOCA-R so that we can measure the accuracy of our tool. This survey will also help to report expert-validated corrections that will further assist software-engineering researchers working on the correction of antipatterns in REST APIs.

In the following, we describe a set of antipatterns and the correction of this antipattern as suggested by our approach in SOCA-R. We ask you to kindly provide your agreement with each correction and, if warranted, explanations for your disagreement.

1. **Breaking Self-descriptiveness**

Problem: REST developers tend to forgo standardized headers, formats, or protocols and use their own customized ones. Anything not being standardized by an official standards body breaks this constraint, and can be considered a case of this antipattern

Example: When your browser retrieves some protected resource's PDF representation, you can see how all of the existing agreements in terms of standards kick in: some HTTP authentication exchange takes place, there might be some caching and/or revalidation, the content-type header sent by the server ("application/pdf") triggers the startup of the PDF viewer registered on your system, and finally you can read the PDF on your screen.

Known Implementations: Every time you invent your own headers, formats, or protocols you break the self-descriptiveness constraint to a certain degree. This antipatterns can be found in various format.

Correction: Remove non-standardized headers in request and response fields returned by the REST APIs providers. Standardized headers list are prepared based on IETF standards [205] .

Agree □□□ Disagree □□□

If Disagree, could you please explain why and, possibly, suggest an alternative correction or that this antipattern should not be corrected at all.

2. Forgetting Hypermedia

Problem: The lack of hypermedia, i.e., not linking resources, hinders the state transition for REST applications. One possible indication of this antipatterns is the absence of URL or links in the resource representation, a connected set of resources, where applications move from one state to the next by following links.

Example:

```
{
   "name": "Alice",
   "links": [ {
      "rel": "self",
      "href": "http://localhost:8080/customer/1"
   } ]
}
```

If "href" is not available in link/location then this is forgetting hypermedia antipatern.

Known Implementations: REST API resource found without ' link' attribute.

Correction: Check the status code of response against each request if successful then add location and link (URL) dynamically if not returned by the REST APIs provider.

Agree □□□ Disagree □□□

If Disagree, could you please explain why and, possibly, suggest an alternative correction or that this antipattern should not be corrected at all.

3. Ignoring Caching

Problem: REST clients and server-side developers tend to avoid caching due to its complexity to implement. The developers ignore caching by setting Cache-Control: no-cache or no-store and by not providing an E-Tag in the response header.

Example:

Response:

Status Code :200

Header: {x-frame-options=[SAMEORIGIN], content-type=[application/json;charset=utf-8], x-rate-limit-remaining=[899], last-modified=[Tue, 13 Jun 2017 23:18:06 GMT], status=[200 OK], x-response-time=[199], date=[Tue, 13 Jun 2017 23:18:06 GMT], x-transaction=[00886c680009e817], pragma=[no-cache], **cache-control=[no-cache, no-store, must-revalidate, pre-check=0, post-check=0],** x-connection-hash=[7036366a9883905f6e9a38d8a0f38562], x-xss-protection=[1; mode=block]

Request :

Header: {cache-control=[no-cache], content-type=[application/xml], connection=[keep-alive], host=[api.twitter.com], accept=[application/xml], get /1.1/users/show.json?screen_name=testjohann http/1.1=[null], user-agent=[Apache CXF 2.7.5], authorization=[**Bearer AAAAAAAAAAAAAAAAAAAAAGbEVgAAAAAAN5fFBEzZiajEU4rS6rahlZIk2VU%3D27uIkvHeAAhVsLSYGG94uOdo8GaO5wh0jjmGt47kGsHyZA4nk1], pragma=[no-cache]}**

Known Implementations: Mostly use no-Cache with out E-tag

Correction: Add dynamically a unique E-Tag for each request and set Cache-Control from no-cache to private or public if not available **.**

Agree □□□ Disagree □□□

If Disagree, could you please explain why and, possibly, suggest an alternative correction or that this antipattern should not be corrected at all.

4. **Ignoring MIME Types**

Problem: The server should represent resources in various formats e.g., XML, JSON, PDF, etc., which may allow clients, developed in diverse languages, a more flexible service consumption. However, server-side developers often intend to have a single representation of resources or rely on their own formats, which limits the resource (or service) accessibility and reusability.

Example: A resource might have a representation in different formats such as XML, JSON, or YAML, for consumption by consumers implemented in Java, JavaScript, and Ruby respectively. Or there might be a “machine-readable” format such as XML in addition to a PDF or JPEG version for humans…

Known Implementations: mostly used JSON.

Correction: Fulfill the client request and represent resources in a format as requested by client by adding Mime type.

Agree □□□ Disagree □□□

If Disagree, could you please explain why and, possibly, suggest an alternative correction or that this antipattern should not be corrected at all.

5. Ignoring Status Code

Problem: Despite of a rich set of well-defined application-level status codes suitable for various contexts, REST developers tend to avoid them, i.e., rely only on common ones, namely 200, 404, and 500, or even different or no status codes. The correct use of status codes from the classes 2xx, 3xx, 4xx, and 5xx helps clients and servers to communicate in a more semantic manner.

Example: Mostly applications treat *all* 2xx codes as success indicators, even if it hasn't been coded to handle the specific code that has been returned. For example Status code '201' should treated as "created" instead of success.

Known Implementations: Many applications that claim to be RESTful return only 200 or 500, or even 200 only (with a failure text contained in the response body)

Correction: Check status code retuned by the server for each request and correct the description of status code or return correct status code as per response.

Agree □□□ Disagree □□□

If Disagree, could you please explain why and, possibly, suggest an alternative correction or that this antipattern should not be corrected at all.

6. Misusing Cookies

Problem: Statelessness is an essential REST principle: session state in the server side is disallowed and any cookies violate Restfulness. Sending keys or tokens in the Set-Cookie or Cookie header field to server-side session is an example of misusing cookies, which concerns both security and privacy.

Example:

Cookies Restful:

Authentication details or 'is logged in' kind a stufflast viewed page or place in application etc.

Cookies Not Restful:

Storing session information

Known Implementations: Most of the REST API providers use resource state or client state and resolve the Misusing cookie antipattern.

Correction if Cookies are used to store the session information of key then remove this from ***'Set-Cookie '*** *field* .

Agree □□□ Disagree □□□

If Disagree, could you please explain why and, possibly, suggest an alternative correction or that this antipattern should not be corrected at all.

…

7. Tunneling Everything Through GET

Problem: Being the most fundamental HTTP method in REST, the GET method retrieves a resource identified by its URI. However, developers may rely on the GET method to perform other kind of actions or operations including creating, deleting, and updating resources. GET is an inappropriate method for any actions other than accessing a resource.

Example: …

Known Implementations: …

Correction: Remove action verbs from URIs of get methods and correct the URIs.

Agree □□□ Disagree □□□

If Disagree, could you please explain why and, possibly, suggest an alternative correction or that this antipattern should not be corrected at all.

…

8.Tunnelling Everything Through POST

Problem: This antipattern is similar to the previous one, except that in addition to the URI, the body of the HTTP POST request may embody operations and parameters to apply on the resource. Developers may depend only on the POST method for sending any types of requests to the server including accessing, updating, or deleting a resource. In general, the proper use of POST is to create a server-side resource.

Example: …

Known Implementations: …

Correction: Remove verbs from URIs and inspect body after getting response and remove verbs.

Agree □□□ Disagree □□□

If Disagree, could you please explain why and, possibly, suggest an alternative correction or that this antipattern should not be corrected at all.

Appendix III: Correction Algorithm for REST Antipatterns

Algorithm 1: Tunneling through GET and Tunneling through POST removed

Input: *request URI, Httpmethod ,request body*

Output: *Tunneling Antipattern remove .*

```
request URI ⟵ GET ,POST ;
keywords[]⟵ "method","action","operation" ;
if keywords contains request-URI
ORCheck-Verb(request-URI) = (TRUE) then
    remove verbs and keywords from URI nodes
end
if keywords contains request-URI
ORCheck-Verb(request-URI) = (TRUE)
OR  keywordscontainsrequestbody
OR CheckVerbs(request-body)=TRUE then
    remove verbs and keywords from Request body and Request URI
end
" Tunneling through GET and POST removed
```

Algorithm 2: Correction Heuristics for Ignoring Cache Antipatterns

Input: *response body, response header , http method*

Output: *Ignoring Status Code antipattern remove .*

```
1 client caching ⟵ request Header.getValue("Cache-control") ;
2 response caching ⟵ "responseheader.getvalue("Cache Control") ;
3 if client cache or server cache ≠ no-cache and no-store then
4 |   responseMetaData.add("cache-control","[private]")
5 |   if responseMetadata.containskey ≠ ETag then
6 |   |   responseMetadata.add(java.util.randomUUID.randomUUID().toString())
7 |   end
8 end
9 "Ignoring Cache Antipatterns Remove "Response Cache Design
  Pattern Detected
```

Algorithm 3: Correction Heuristics for Breaking Self Descriptiveness Antipatterns

Input: *request header,response header*

Output: *Breaking Self Descriptiveness antipattern remove .*

```
1 request header ⟵ response.getRequest().getMetaData();
2 response header ⟵response.getMetaData();
3 if (!requestHeaders.contains(header) and
    !header.contains("http/1.1")) then
4 |  "Remove non standard request "
5 end
6 if (!responseHeaders.contains(header)) then
7 |  "Remove non standard response "
8 end
9 "Breaking Self Descriptiveness antipattern remove "
```

Algorithm 4: Correction Heuristic of Misusing Cookie Antipattern Removal

Input: *cookie,set-cookie*

Output: *Misusing Cookie antipattern remove .*

```
1  if response.getMetaData().containsKey("cookie") then
2  |   response.getMetaData().remove("cookie")
3  |   if response.getMetaData().containsKey("set-cookie") then
4  |   |   response.getMetaData().remove("set-cookie")
5  |   end
6  end
7  if response.getRequest().getMetaData().containsKey("cookie") then
8  |   response.getRequest().getMetaData().remove("cookie")
9  |   if response.getRequest().getMetaData().containsKey("set-cookie")
   |    then
10 |   |   response.getRequest().getMetaData().remove("set-cookie")
11 |   end
12 end
13  Misusing Cookie Antipattern remove
```

Algorithm 5: Correction Heuristic of Ignoring Status Code Antipattern Removal

Input: *Response body, Response Header, http-method*

Output: *Ignoring Status-code antipattern remove.*

```
1  http-method ⟵ response.getRequest().getMethod();
2  Response body ⟵ extractStatusTextFromBody(response.getBody()
3  Response header ⟵
     extractStatusCodeFromBody(response.getstatuscode()
4  int code = response.getStatus();
5  if (s.getCode() != code) then
6  |   response.setStatus(s.getCode()); code = response.getStatus();
7  |   //add Status code and description dynamically
8  |   ("Response Code Changed...." +
   |    response.addStatus()+response.addCode());
9  end
10 "Code received and description received matched"
11  Ignoring Status Code Antipattern remove
```