

Extending Maintainability Analysis Beyond Code Smells

DISSERTATION FOR THE AWARD OF THE DOCTORAL
DIPLOMA
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

2019

Tushar Sharma
Department of Management Science and Technology
Athens University of Economics and Business



Department of Management Science and Technology
Athens University of Economics and Business
Email: tushar@aueb.gr

Copyright 2019 Tushar Sharma

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.



Supervised by Professor **Diomidis Spinellis**



Dedicated to my father
who could not see this thesis completed





Contents

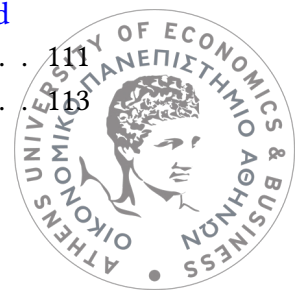
1	Introduction	1
1.1	Context	1
1.2	Problem Statement	2
1.3	Proposed Solution and Contributions	4
1.4	Research method	6
1.5	Thesis outline	6
2	Related Work	8
2.1	Introduction	8
2.2	Method	9
2.2.1	Research objectives and questions	10
2.2.2	Literature search protocol	10
2.2.2.1	Literature search – Phase 1	11
2.2.2.2	Literature search – Phase 2	12
2.2.2.3	Literature search – Phase 3	12
2.3	Results and Discussion	13
2.3.1	LR-RQ1: What is the definition of a software smell?	14
2.3.1.1	LR-RQ1.1: What are the defining characteristics of a software smell?	14
2.3.1.2	LR-RQ1.2: What are the types of smells?	14
2.3.1.3	LR-RQ1.3: How are the smells classified?	19
2.3.1.4	LR-RQ1.4: Are smells and antipatterns considered synonyms?	20
2.3.2	LR-RQ2: How do smells get introduced in software systems?	21
2.3.3	LR-RQ3: How do smells affect the software development processes, artifacts, or people?	23
2.3.4	LR-RQ4: How do smells get detected?	24
2.3.4.1	Machine learning techniques on source code	28
2.3.5	LR-RQ5: What are the open research questions?	31
2.4	Conclusions	39
3	Methodology	41
3.1	Research Objectives	41



3.1.1	Maintainability Analysis for Production Source Code	41
3.1.2	Detecting Smells using Deep Learning	43
3.1.3	Maintainability Analysis for Configuration Code	45
3.1.4	Maintainability Analysis for Database Code	47
3.2	Theoretical Background	48
3.2.1	Code Smells	48
3.2.1.1	Architecture Smells	49
3.2.1.2	Design Smells	49
3.2.1.3	Implementation Smells	49
3.2.2	Exploring Deep Learning-based Solution for Smell Detection	50
3.2.2.1	Challenges in Applying Deep Learning on Source Code	51
3.2.2.2	Selection of Smells	53
3.2.3	Configuration Smells	53
3.2.3.1	Implementation Configuration Smells	54
3.2.3.2	Design Configuration Smells	56
3.2.4	Database Smells	58
4	Implementation	62
4.1	Analyzing Production Code for Quantitative Maintainability Assessment	62
4.1.1	Mining C# Repositories	62
4.1.2	Analyzing C# Repositories Using Designite	63
4.1.2.1	Architecture	64
4.1.2.2	Detection Mechanism for Supported Architecture Smells	64
4.1.2.3	Detection Mechanism for Supported Design Smells	66
4.1.2.4	Detection Mechanism for Supported Implementation Smells	68
4.1.2.5	Evaluation	69
4.2	Detecting Smells using Deep Learning	71
4.2.1	Data Generation and Curation	71
4.2.1.1	Downloading Repositories	71
4.2.1.2	Smell Detection	72
4.2.1.3	Splitting Code Fragments	72
4.2.1.4	Generating Training and Evaluation Data	72
4.2.1.5	Tokenizing Learning Data	72
4.2.1.6	Data Preparation	73
4.2.2	Architecture of Deep Learning Models	74
4.2.2.1	CNN Model	74
4.2.2.2	RNN Model	76
4.2.3	Hardware Specification	77
4.3	Analyzing Configuration Code for Quantitative Maintainability Assessment	78
4.3.1	Selecting and Downloading Puppet repositories	78
4.3.2	Design Configuration Smells – Detection Strategies	79
4.4	Analyzing Database Code for Maintainability Assessment	81



4.4.1	Mining Repositories	81
4.4.1.1	Selecting Industrial Repositories	81
4.4.1.2	Selecting Open-source Repositories	81
4.4.1.3	Extracting SQL Statements	82
4.4.1.4	Analyzing and Detecting Smells	82
4.4.2	DbDeo and Detection Strategies for Database Smells	82
4.4.3	Accuracy of DbDeo	84
4.4.3.1	Accuracy of the SQL Statements Extraction	84
4.4.3.2	Accuracy of Smell Detection	85
5	Results and Discussion	86
5.1	Results of Maintainability Analysis on Production Code	86
5.1.1	P-RQ1. What is the distribution of implementation, design, and architecture smells in C# code?	86
5.1.2	P-RQ2. Do the detected smell instances belonging to different granularities correlate?	89
5.1.3	P-RQ3. Is the principle of coexistence applicable to smells in C# projects?	90
5.1.4	P-RQ4. Does smell density depend on the size of the C# repository?	92
5.1.5	P-RQ5. Are architecture smells collocated with design smells?	93
5.1.6	P-RQ6. Can the refactoring of design smells lead to fewer architecture smells?	97
5.1.7	Discussion and Implications	101
5.1.7.1	Discussion	101
5.1.7.2	Secondary Uses of this Work	102
5.2	Results of Detecting Smells using Deep Learning	103
5.2.1	D-RQ1. Is it possible to use deep learning methods to detect code smells? If yes, which deep learning method performs superior?	103
5.2.1.1	D-RQ1.H1. It is feasible to detect smells using deep learning methods.	105
5.2.1.2	D-RQ1.H2. CNN-2D performs better than CNN-1D in the context of detecting smells.	107
5.2.1.3	D-RQ1.H3. RNN model performs better than CNN models in the smell detection context.	107
5.2.2	D-RQ2. Is transfer-learning feasible in the context of detecting smells? If yes, which deep learning model exhibits superior performance in detecting smells when applied in transfer-learning setting?	108
5.2.2.1	D-RQ2.H1. It is feasible to apply transfer-learning in the context of code smells detection.	110
5.2.2.2	D-RQ2.H2. Transfer-learning performs inferior compared to direct learning.	111
5.2.3	Discussion	113

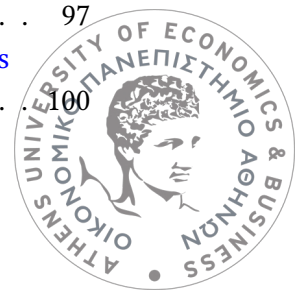


5.2.3.1	Is there any silver-bullet?	113
5.2.3.2	Performance comparison with baseline	113
5.2.3.3	Poor performance in detecting a design smell	114
5.2.3.4	Trading performance with training-time	115
5.3	Results of Maintainability Analysis on Configuration Code	116
5.3.1	C-RQ1. What is the distribution of maintainability smells in configuration code?	116
5.3.2	C-RQ2. What is the relationship between the occurrence of design configuration smells and implementation configuration smells? . . .	117
5.3.3	C-RQ3. Is the principle of coexistence applicable to smells in configuration projects?	118
5.3.4	C-RQ4. Does smell density depend on the size of the configuration project?	120
5.3.5	Discussion	121
5.4	Results of Maintainability Analysis on Database Code	122
5.4.1	Developers' Survey on Database Smells	122
5.4.2	DB-RQ1. What are the occurrence patterns of database smells? . . .	124
5.4.3	DB-RQ2. Does the size of the project or the database play a role in smell density?	125
5.4.4	DB-RQ3. Does the nature of code (type of the application, or usage of ORM frameworks) affect the smell density?	126
5.4.5	DB-RQ4. What is the degree of co-occurrence among database smells?	127
5.4.6	Discussion	128
5.4.6.1	Qualitative Analysis of the Results	128
5.5	Threats to Validity	130
5.5.1	Construct Validity	130
5.5.2	Internal Validity	131
5.5.3	External Validity	131
6	Conclusions and Future Work	133
6.1	Summary of the Results	133
6.2	Contributions of the Thesis	135
6.3	Future Work	137
	Bibliography	140



List of Figures

2.1	Overview of the study; a number in brackets shows the number of associated references	13
2.2	A layered overview of smell detection methods. Each detection method starts from the code (or other source artifact) and goes through various steps to detect smells. The direction of the arrows shows the flow direction and annotations on the arrows show the detection method (first part) and the step number (second part).	25
2.3	A recorded smell could be a false-positive instance, a smell that is not a quality problem, or a definite quality problem.	32
2.4	The number of studies detecting a specific smell	37
3.1	Overview of the maintainability analysis study on C# code	43
3.2	Overview of the Proposed Method	45
3.3	Overview of the maintainability analysis study on configuration (Puppet) code	46
3.4	Overview of the maintainability analysis study on database schema code	48
3.5	An annotated Puppet example with all the cataloged implementation configuration smells	54
4.1	Presentation of identified smells in Designite	64
4.2	Architecture of the tool	65
4.3	Tokens generated by Tokenizer for an example	73
4.4	Architecture of employed CNN	75
4.5	Architecture of employed RNN	76
5.1	Scatter plots showing co-occurrence between smells in two granularities	90
5.2	Correlation between individual architecture and design smells	91
5.3	Average co-occurrence (intra-category) for architecture smells	92
5.4	Average co-occurrence (intra-category) for design smells	92
5.5	Average co-occurrence (intra-category) for implementation smells	93
5.6	Smell density for implementation, design, and architecture smells against lines of code	94
5.7	Collocation analysis between architecture and design smells	97
5.8	Removed architecture smells (in percentages) after simulating design smells refactoring	100



5.9	Scatter plots of the performance (F1) exhibit by the considered deep learning models along with their corresponding trendline	104
5.10	Boxplots of the performance (F1) exhibit by the considered deep learning models for all the four smells	105
5.11	Comparative performance of the deep learning models for each considered smell	106
5.12	Scatter plots for each model and for each considered smell comparing F1 of direct-learning and transfer-learning along with corresponding trendline . .	109
5.13	Comparative performance of the deep learning models for each considered smell in transfer-learning settings	110
5.14	Comparison of performance of the deep learning models between direct-learning (DL) and transfer-learning (TL) settings	111
5.15	Co-occurrence between implementation and design configuration smells by (a) volume and by (b) existence	118
5.16	Average co-occurrence (intra-category) for implementation and design configuration smells	119
5.17	Smell density for (a) implementation configuration smells and (b) design configuration smells against lines of code	120
5.18	Experience of respondents in terms of number of years as well as the number of database applications developed by them	122
5.19	Respondents' perspective of considered database smells	123
5.20	Average smell density of different types of applications (left) and projects using ORM frameworks and rest of the projects (right)	127
5.21	Average co-occurrence among database smells	128



List of Tables

2.1	Studies selected in the Phase 1	11
2.2	Studies selected in the Phase 2	12
2.3	Types of smells	15
2.4	Common code smells	15
2.4	Common code smells	16
2.4	Common code smells	17
2.5	Actor-based Classification of Smells Causes	23
2.6	Impact of Smells	24
2.7	Smell Detection Methods and Corresponding References	26
2.7	Smell Detection Methods and Corresponding References	27
2.7	Smell Detection Methods and Corresponding References	28
2.8	Smell Detection Methods and supported smells	34
2.8	Smell Detection Methods and supported smells	35
2.8	Smell Detection Methods and supported smells	36
3.1	Description of Detected Design Smells	50
3.2	Description of Detected Implementation Smells and Their Distribution	51
3.3	Mapping Between Implementation Configuration Smells and Corresponding Best Practices	55
4.1	Characteristics of the Analyzed Repositories	63
4.2	Results of Manual Validation	70
4.3	Number of samples in each step of preparing input data	73
4.4	Chosen values of hyper-parameters for the CNN model	76
4.5	Chosen values of hyper-parameters for the RNN model	77
4.6	Characteristics of the Downloaded Repositories	78
4.7	Characteristics of the analyzed industrial (I) as well as open-source (OSS) repositories	83
4.8	Performance of the SQL extraction process	85
4.9	Detected smells and identified false-positives	85
5.1	Distribution of Implementation Smells	87
5.2	Number of detected instances and smell density (per KLOC) of design smells in the analyzed repositories	88



5.3	Number of detected instances and smell density (per KLOC) of architecture smells in the analyzed repositories	88
5.4	Contingency matrix for a design and architecture smell	95
5.5	Architecture smell instances detected before and after the refactoring simulation for design smells	100
5.6	Number of positive (P) and negative (N) samples used for training and evaluation for RQ1	103
5.7	Performance of all three models with configuration corresponding to the optimal performance. L refers to deep learning layers, F refers to number of filters, K refers to kernel size, MPW refers to maximum pooling window size, ED refers to embedding dimension, LSTM refers to number of LSTM units, and E refers to number of epochs.	106
5.8	Performance (F1) comparison of RNN with CNN-1D and CNN-2D	107
5.9	Positive (P) and negative (N) number of samples used for training and evaluation for RQ2	110
5.10	Performance of all three models with configuration corresponding to the optimal performance. L refers to deep learning layers, F refers to number of filters, K refers to kernel size, MPW refers to maximum pooling window size, ED refers to embedding dimension, LSTM refers to number of LSTM units, and E refers to number of epochs.	111
5.11	Difference in ratio (in percent) of positive and negative evaluation samples in RQ2 compared to sample ratio in RQ1	112
5.12	Comparison of performance (Precision, Recall, and F1) with a random classifier (RC) following the training set frequencies or responding always indicating a smell	114
5.13	Average training-time taken by the models to train a single epoch in seconds	115
5.14	Distribution of Detected Implementation and Design Configuration Smells .	117
5.15	Results of Correlation Analysis	119
5.16	Results of Correlation Analysis	121
5.17	Occurrences of database schema smells for industry (I) as well as open-source (OSS) repositories	124



Acknowledgements

गुरु ब्रम्हा गुरु विष्णुः गुरु देवो महेश्वरः ।
गुरुः साक्षात् परंब्रह्म तस्मै श्री गुरवे नमः ॥

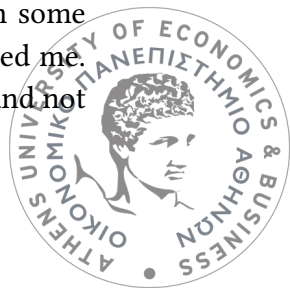
—by Veda Vyāsa in *Skanda Purana*

Meaning: My Guru (teacher) is the representative of *Brahma*, *Vishnu*, and *Shiva*. He creates, sustains knowledge and destroys the weeds of ignorance. I offer my obeisance to my Guru.

The above *shloka* precisely capture my thoughts about my teachers Prof. Diomidis Spinellis and Prof. Panos Louridas. Prof. Spinellis put a lot of faith in me and accepted me as his student. He is a great mentor, researcher, developer, and administrator. Despite his very busy schedule, he always offered his help and advice related to research, teaching, or even non-technical aspects coming from his abundant knowledge and experience. He doesn't forget to pat the back on our achievements but also stand tall and support his students on each tumble (such as a paper rejection). Behind his technical avatar, he is a person with very pure heart who is always ready to offer his help on any matter. From the beginning, whether I required his help with the initial paperwork to come to Greece, advice on finding good restaurants or nice islands to visit, finding a tax consultant, or discussing the trade-offs between academia and industry to choose the future course, his unparalleled support made my migration to Greece and staying in Athens possible and a joyful experience. I am indebted to him for the great amount of energy and time he invested in supervising me.

Prof. Panos Louridas — well, self declared non-statistician who in reality knows statistics probably more than the rest of the lab combined. If any of us dealing with any mathematics, machine learning, or algorithms, he is our go-to person. The first thing you will notice about him is his smile and energy; he is kind and always ready to help. He is a ground to earth person who reveals the richness of his copious knowledge when one works with him. He pays attention to the details, seeks perfection, and makes you work harder. I learned a lot of things from him.

Marios is the first Greek friend I got. When I landed in Athens first time with some worries of unknown future, this warm-hearted and friendly person's smile welcomed me. He not only came to pick me up from the airport (which was officially not required and not



expected), and dropped me to my airbnb but also helped me with the Greek supermarkets to make sure a smooth settling down for me. Getting resident permit in Greece is a lengthy and laborious affair; without Marios's help I don't think I could have done it. He spent countless days and went the extra mile always to facilitate me fulfilling the requirements of staying in Greece. We also enjoy our technical discussions on various topics — whether it is about our ongoing work, some random ideas that may lead to interesting papers, or even startup ideas. I am sure TU Delft is giving more wings to his career.

I can't forget our erstwhile lab in the main building of AUEB where a *Kalimera* from Maria used to welcome me every morning. Maria is a hardworking, determined, and cheerful person. I remember countless discussions with her not limited to technical topics that used to start during our lunch break but frequently used to spill over even when we come back to the lab. I miss the delicious *moustokolora* and cake that her mother cooks for her and she generously shared with us.

Antonis and Stefanos — my beer buddies. Both are the living alarms — Stefanos for lunch and Antonis for beer; well, for Antonis it is a little bit exaggeration but not at all for Stefanos :). We are like brothers — we shared many activities such as having lunch, drinking beers, sharing thoughts, admiring ;) and laughing on each other and literally grew up for three years together. We did many trips together and I thoroughly enjoyed their company. Both of them value friendship and they know how to earn and keep people as friends. They go to any extreme to help their friends in need and I am blessed to be their friend. Antonis is mature and sensible while Stefanos gained the wisdom tooth recently when he entered into his thirties.

When you organize a big event such as SATTOSE¹, you need reliable and dependable shoulders to share your load of thousand things that you need to put together. Vasiliki gladly accepted the challenge and put tremendous effort to make the event successful; she is the single biggest reason behind the flawless execution of the event. She is always ready to listen, share her honest opinions, and most importantly sacrifice her time to help anybody in need. She has reviewed many of my papers and this thesis and helped me improve the text and my writing. She put a lot of effort in whatever she takes up; I realized this yet again when we were working on using machine learning techniques on smell detection. I wish her more ELKE-less time.

SATTOSE reminds me of another person, Alexandra, who volunteered herself for the event and contributed to the event even when she was doing her summer internship. She put together the initial structure and contributed generously towards the DesigniteJava project. We developed a connection and friendship beyond technical discussions as the frequency of our thoughts strikes a harmonious note.

What! Stereo is down!! You know, you need to contact Konstantinos — the new young lad who is carrying the responsibilities of maintaining Stereo. He is smart, patient, and multi-tasker. Beside ensuring that Stereo always on and hot, he knows how to use a hot oven to produce delicacies.

Singular Logic, as my industrial host in Athens, supported well in the entire journey.

¹<http://sattose.org/2018>



Specifically, Matina was always motivating and supporting. She not only shaped and fine-tuned the technical ideas (especially when I was analyzing database schemas) but also provided her unparalleled support to organizational and operational aspects.

My industrial host in Amsterdam, *i.e.*, SIG not only hosted me in their office multiple times but also mentored my thoughts and supported my research. Specifically, both Joost and Magiel gratified me with their ample experience and knowledge apart from providing exceptional operational support from SENECA perspective.

The support of the Department of Management Science and Technology at AUEB along with all the professors, teaching and support staff has contributed to my journey. Specifically, I would like to convey my sincere thanks to Prof. Damianos Chatziantoniou for reviewing an earlier draft of my database schema quality paper and providing improvement suggestions. I would also like to thank Anna Klouvatou from business administration to ensure processing my payments on time.

My sincere thanks to Prof. Paramvir Singh from NIT Jalandhar who thoroughly supported me with my work on architecture smells. I wish him success for his new academic journey at the University of Auckland.

My stay in Greece would have not been possible if I didn't get fully funded by the SENECA project. I am sincerely thankful to the Marie Skłodowska-Curie Innovative Training Networks (ITN-EID) under which the SENECA project was funded (grant agreement number 642954). Also, I would like to convey my warm thanks to all the partners that participated in the project.

The support that I received from my family whether it was the bold decision to leave a settled life and move to Greece, or have patience when I was burning mid-night oil gave me strength to keep going on.

Last but not the least, I offer a bow to all the Greeks in general who accepted me socially and made my three years a memorable life experience. I can recall countless people such as students from my SEIP class, Vangelis (my landlord), our neighbors as well as unknown Greeks who directly or indirectly made my stay easier and joyful.

- Tushar Sharma



Summary

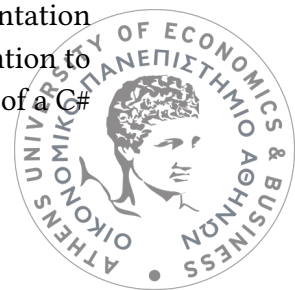
Code smells indicate the presence of quality problems impacting many facets of software quality such as maintainability, reliability, and testability. The presence of an excessive number of smells in a software system makes it hard to maintain and evolve.

Software engineering researchers have carried out many empirical and mining studies on code smells impacting various dimensions of software development. Our first aim in this thesis is to understand the characteristics of code smells, such as their occurrence frequency, and relationships such as correlation and collocation among smells arising at different granularities. We aim to realize the experiment with an extended scale (*i.e.*, number of analyzed subject systems) and breadth (*i.e.*, mining a large variety of smells).

The software engineering community has proposed various methods to detect smells. Machine learning techniques offer a promising alternative to deterministic smell detection methods and provide the grounds for applying transfer-learning from one programming language to another. We aim to perform an exploratory study to investigate the feasibility of detecting smells using deep learning methods without carrying out extensive feature engineering. We would also like to explore whether transfer-learning can be employed in the smell detection context.

Apart from the production source code, other sub-domains of software such as configuration code in Infrastructure as Code (IaC) paradigm and database code are also prone to maintainability issues. Our next goal is to propose a method to identify quality issues in configuration code and carry out a maintainability analysis. We also would like to explore the relationships between different kinds of smells at inter- as well as intra-category. Similarly, we would like to propose a mechanism to collate, evaluate, and detect smells that may arise in database schema design. We would like to propose a method to investigate code quality of embedded SQL statements, understand the impact of quality issues in connection with properties of database and production code, and pinpoint areas where improvement in tools, processes, or methods could be proposed to keep the database quality high.

We perform a large-scale empirical study to analyze production code written in C# from maintainability perspective. We mine seven architecture, 19 design, 11 implementation smells from a large set of 3,209 open-source repositories containing more than 83 million lines of code. We find that *cyclic dependency*, *cyclically-dependent modularization*, and *magic number* are the most frequently occurring architecture, design, and implementation smells respectively. This observation may prompt developers to pay additional attention to avoid frequently occurring smells. Our analysis observes that smell density and size of a C#

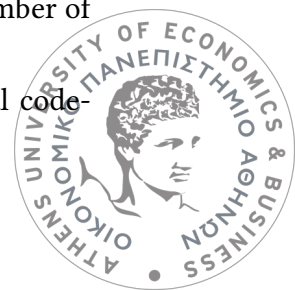


project show a weak correlation. The co-occurrence analysis shows that the architecture smells exhibit a strong positive correlation with design smells. This implies that a project containing a high number of design smells would also exhibit a high number of architecture smells and vice-versa. We also perform fine-grain correlation between individual smell pairs using Spearman correlation analysis. The results of individual pair-wise correlation analysis indicate that design and architecture smells exhibit a non-monotonic relationship. The collocation analysis reveals that apart from a few selected smell pairs, architecture and design smells do not collocate with each other. We also explore the potential influence of design smells refactoring on architecture smells. Our analysis shows that up to one-third of architecture smells (in case of *god component*) may get removed if we refactor all detected design smells in the component. However, a significant number of architecture smells persist even after all the smells at design granularity were refactored. This result emphasizes the need to carry out smell detection and refactoring at all source-code granularities.

In our exploration with deep learning techniques to identify smells, we develop a set of tools (such as Designite, CodeSplit, and Tokenizer) and put together an experimental setup to detect smells, generate code fragments, and tokenize them to feed into our deep learning models (specifically, Convolution Neural Network and Recurrent Neural Network). We perform the experiments with various combination of hyper-parameters for each of the model. Our result establishes that deep learning methods (specifically CNN and RNN in our case) can be used for smell detection though the performance of individual models varies significantly. We find that there is no clear winner between 1-D and 2-D convolution neural networks; CNN-1D performs better for smells *empty catch block* and *multifaceted abstraction* while CNN-2D performs superior than its one-dimensional counterpart for smells *complex method* and *magic number*. We also observe that performance of the deep learning models is smell-specific. Our experiment with applying transfer-learning proves the feasibility of practicing transfer-learning in the context of smells detection especially for the implementation smell.

We extend the maintainability analysis to configuration code. We propose a catalog of 13 implementation and 11 design configuration smells based on commonly known best practices. We analyze 4,621 Puppet repositories containing 142,662 Puppet files and more than 8.9 million lines of code using Puppeteer — a configuration smell detection tool that we developed. Our analysis finds that the developers of Puppet repositories either do not introduce code-clones at all or they do it massively. The inter-category correlation analysis for configuration smells shows a strong correlation between smells belonging to different categories. Design configuration smells show 9% higher average co-occurrence among themselves than the implementation configuration smells. This observation affirms the belief that one wrong or non-optimal design decision introduces many quality issues and therefore suggests the developers to take design decisions critically and diligently. Design configuration smell density shows negative correlation whereas implementation configuration smell density exhibits no correlation with the size of a project. It shows that the number of design configuration smells decrease as the size of the configuration code increases.

Further, we carry out a comparative study between open-source and industrial code-



base from database schema quality perspective. The study investigates relational database schema smells and its relationships with application and database characteristics. We present a catalog of 13 database schema smells based on commonly known best practices to design databases. We carry out a survey to understand developers' perspective on database schema smells. We download 16,052 open-source and acquire 840 industrial repositories, select total 2,925 repositories containing SQL statements, analyze more than 629 million lines of code, extract more than 393 thousand SQL statements, and detect more than 66 thousand instances of database schema smells. We observe that the smell *index abuse* occurs most frequently in database code. We also find that some smells such as adjacency list show significantly higher proneness to occur in industrial projects compared to open-source projects. Our analysis shows that the size of the host application has no impact on the density of database smells; however, smell density shows a positive correlation with the size of the database whereas application type (Desktop, Mobile, or Web) has no significant impact on database smell density. Finally, the use of an ORM framework does not help developers to avoid database schema smells.

In summary, the thesis offers contributions to both research and practice aspects. From the research perspective, the thesis proposes methods to carry out large-scale (both in terms of number of subject systems and kinds of code smells detected) empirical studies for not only production source code but also for configuration and database code. The methods aim to understand characteristics of code smells at different granularities and subfields of software engineering as well as to explore interesting relationships among the smells. In addition, the thesis presents a detailed mechanism to show the feasibility of detecting code smells using deep learning methods. Also, the method applies transfer-learning to showcase that a deep learning classifier trained from a programming language can be used to identify smelly code fragments belonging to another programming language. Apart from research-oriented contributions, the thesis also adds contributions towards software engineering practice. The thesis offers a set of tools: *Designite*—to detect a wide variety of implementation, design, and architecture smells in C# source code, *Puppeteer*—to identify configuration smells in Puppet code, *DbDeo*—to identify database schema smells in embedded SQL statements. Practitioners may use the various features offered by the tools to identify maintainability issues in not only their production source code but also in their database and configuration code to reduce technical debt.



Chapter 1

Introduction

| *Context is the water for the fishes of our ideas.*

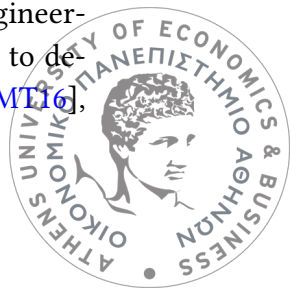
In this chapter, we provide the context of the work presented in this thesis, the problem statement, our proposed solution and a summary of contributions.

1.1 Context

Kent Beck coined the term “code smell” in the landmark book of refactoring [Fow99] and defined it as “*certain structures in the code that suggest (or sometimes scream) for refactoring*”. Code smells indicate the presence of quality problems impacting many facets of quality [SS18] of a software system [Fow99, SSS14]. The manifestation of an excessive number of smells in a software system makes it hard to maintain and evolve.

The impact of code smells on software development is multi-dimensional. Researchers have explored and discussed the impact of the high number of code smells on specific aspects of software development extensively. These aspects are associated with one of the three dimensions of software development: *software product*, *processes*, and *people*. From the software product perspective, aspects such as maintainability [BQO⁺12, MY12, YM13a, Yam14], reliability [JGHK13, HZBS14, ZSSS11, BQO⁺14], change proneness [OCBZ09, KDPG09] and testability [SDPAG13] have been investigated in considerable depth. In addition, source code mining studies have examined empirical aspects associated with developers and code smells [BDLDP⁺15, SCY⁺16].

Detecting smells is the first step towards identifying quality issues that lays the ground work for improving maintainability by applying appropriate refactorings. From the research perspective, detected smells form the basis of empirical studies. The software engineering community has proposed various methods to detect smells. Common methods to detect smells are metric-based [Mar05, VMDP14], rules/heuristic-based [MGDM10, SMT16],



history-based [PBDP⁺15], machine learning-based [MAB⁺12b, KVGS09], and optimization-based smell detection [OKKI15].

Apart from the production code, the metaphor of code smells could be extended to sub-domains of software systems such as configuration system and databases. Configuration code written in languages such as Puppet [Pup18] and Chef [Che18] may also become unmaintainable if the changes to configuration code are made without diligence and care. Similarly, database code is also prone to smells. Typically, the use of a database in a software system manifests itself as a series of DDL (Data Definition Language — e.g., CREATE TABLE) or DML (Data Manipulation Language — e.g., SELECT) SQL statements. Similar to code, avoiding best practices of the domain may lead to smells in these SQL statements. In this context, Bill Karwin [Kar10] documents a catalog of database anti-patterns. Therefore, production code and design quality practices must be adopted to similar sub-domains of software systems including configuration and database systems.

1.2 Problem Statement



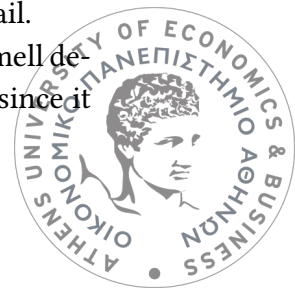
Goal of the thesis

The goal of this thesis is to enrich the existing body of knowledge about code smells by carrying out empirical studies on production source code, by investigating the feasibility of automating smell detection through deep learning techniques, and by extending the maintainability analysis to other sub-domains of software systems.

Software engineering researchers have carried out many empirical and mining studies on code smells impacting various facets of software development or developers. However, we observe that existing mining studies on smells lack *scale i.e.*, they rely on a limited number of subject systems analyzed for the study. The majority of the studies analyze a handful of subject systems (< 10). Generalizing a theory based on a few subject systems presents a considerable threat to validity. Also, existing mining studies do poorly with respect to *breadth* of the experiment *i.e.*, the types of smells analyzed in a study. Most of the existing mining studies consider a small subset of known smells. This under-analysis makes a mining study incomplete or even incorrect. Finally, most of the mining studies on code smells are performed on the Java programming language. The under-representation of other programming languages makes us wonder whether the results of the existing mining studies are applicable in other similar languages.

Along the similar lines, despite the presence of a large body of work for smell detection, the existing research has supported detection of mainly implementation and some design smells. The research on architecture smells and their detection is still in a budding stage [GPem09, SSS16], and requires a serious attention from the community, given their importance and impact on the quality of software systems. The relationship (such as correlation and collocation) among smells at various granularities has not been explored in detail.

Carrying forward the discussion about smell detection, creating a deterministic smell detection tool for a specific programming language is an expensive and arduous task since it



requires source code analysis starting from parsing, symbol resolution, intermediate model preparation, and applying an appropriate mechanism (such as heuristics and metrics) on the model. Machine learning techniques offer promising alternatives to deterministic solutions as they not only have the potential to bring subjectivity, that in turn improves effectiveness, in the smell detection but also provide the grounds for transferring results from one problem to another. In particular, transfer-learning refers to the technique where a learning algorithm exploits the commonalities between different learning tasks to enable knowledge transfer across the tasks [BCV13]. In this context, it would be interesting to explore the possibility of leveraging the availability of tools and data related to code smell detection in a programming language in order to train machine learning models that address the same problem on another language. The cross-application of a machine learning model could provide opportunities for detecting smells without actually developing a language-specific smell detection tool from scratch.

Apart from the production source code, other sub-domains of software such as configuration code in the Infrastructure as Code (IaC) paradigm and database code are also prone to maintainability issues. Infrastructure as Code (IaC) [HF10] is the practice of specifying computing system configurations through code, automating system deployment, and managing the system configurations through production software engineering methods. Configuration code written in Puppet [Pup18], or Chef [Che18], may also become unmaintainable if the changes to configuration code are made without diligence and care. In a recent study, Jiang et al. [JA15] argued that configuration code must be treated as production code due to the characteristics and maintenance needs of the configuration code. Therefore, production code and design quality practices must be adopted to write and maintain high quality configuration code.

Databases are an integral element of enterprise applications. The effective use of database affects vital quality parameters, such as performance and maintainability, of these applications. Similar to production code, the SQL statements may also indicate smells. Bill Karwin [Kar10] documents a catalog of database anti-patterns to reveal many of the practices affecting database quality. However, their presence in software systems and their relationships with other software artifacts have not been explored yet.



Research goals

With the above background, we define the following research goals for the thesis.

- Understand characteristics of code smells and relationships such as frequency, correlation, and collocation among smells arising at different granularities by extending the scale (*i.e.*, number of analyzed subject systems) and breadth (*i.e.*, mining a large variety of smells) of the mining study.
- Perform an exploratory study to investigate the feasibility of detecting smells using deep learning methods without carrying out extensive feature engineering. Explore whether transfer-learning can be employed in the smell detection context.



- Propose a method to identify quality issues in configuration code and carry out a maintainability analysis for code written in the Infrastructure as Code paradigm. In addition, explore the relationships among different kinds of smells at inter- as well as intra-category.
- Explore a mechanism to collate, evaluate, and detect smells that may arise in database schema design. Offer a method to investigate code quality of embedded SQL statements, understand the impact of quality issues in connection with properties of database and production code, and pinpoint areas where improvement in tools, processes, or methods could be proposed to keep the database quality high.

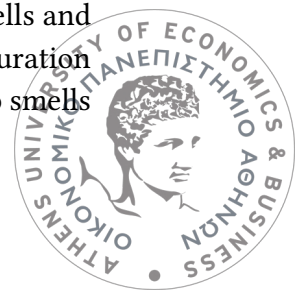
1.3 Proposed Solution and Contributions

The thesis fulfills the first goal by carrying out a large scale empirical study to mine code smells in C# projects. The study reveals fundamental, yet interesting, characteristics of code smells in C# projects. These characteristics include — frequently occurring smells, inter-category and intra-category correlation between design and implementation smells, and the relationship of smell density with lines of code in each repository. Smell density is a normalized metric that represents the average number of smells identified per thousand lines of code.

To study architecture smells in production source code we automate the detection of seven architecture smells in our tool Designite. We analyze a large set of repositories and infer relationships among the smells belonging to two granularities (architecture and design) through correlation and collocation analysis. In particular, the analysis explores whether there are specific design smells that may act as indicators for specific architecture smells.

To fulfill the second goal, we develop a set of tools (such as Designite, CodeSplit, and Tokenizer) and put together an experimental setup to detect smells, generate code fragments, and tokenize them to feed into our deep learning models (specifically, Convolution Neural Networks and Recurrent Neural Networks). We perform the experiments with various combinations of hyper-parameters for each of the models. With the above setup, we first experiment to investigate the feasibility of employing deep learning to detect smells; we keep both our training and evaluation samples from C#. To show the feasibility of transfer-learning, we replace the evaluation samples written in C# with Java samples and document the performance of our models.

For the next goal, we perform a quality analysis of configuration code and explore the distribution of configuration smells. *Configuration smells are the characteristics of a configuration program or script that violate the recommended best practices and potentially affect the program's quality in a negative way.* Refer to Section 3.2.3 for the detailed catalog. We investigate the relationship between the occurrence of design configuration smells and implementation configuration smells. Other related dimensions concerning configuration code quality that we explore are whether the principle of coexistence is applicable to smells

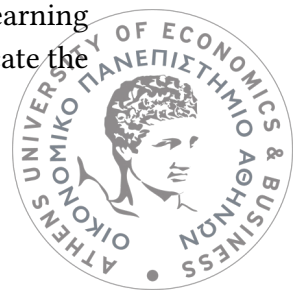


in configuration projects and the relationship between smell density and the size of the configuration project.

Further, towards the last goal defined in this work, we mine database smells in production-quality systems including both industrial as well as the open-source software. *Database smells are the characteristics of database code (either DDL or DML SQL statements), database system, or stored data that indicate violation of the recommended best practices and potentially affect the quality of the software system in a negative way.* You may refer to Section 3.2.4 for a detailed catalog. We analyze SQL statements to measure schema quality of relational databases with a focus on performance and maintainability quality attributes. Specifically, we explore occurrence patterns of database schema smells and figure out the degree of co-occurrence among schema smells. We also study the factors that affect the density of database smells and compare these factors between industrial and open-source projects.

In summary, the thesis offers the following contributions:

- A method to carry out a large-scale empirical study to mine smells from a large number of repositories and to study smells' characteristics in relation with project properties.
- A method to perform correlation and collocation analysis among smells belonging to different granularities.
- A method to explore the feasibility of employing deep learning models in detecting code smells and to investigate whether transfer-learning can be applied in the context of smell detection.
- A method to prepare a catalog of configuration smells and perform an empirical study on quality issues and their characteristics in the IaC paradigm.
- A method to investigate the code quality of embedded SQL statements by identifying database schema smells in a large set of repositories and study the impact of smells on project properties.
- A set of tools to identify code smells. This includes Designite (for C#) to identify architecture, design, and implementation smells, Puppeteer to identify configuration code smells in Puppet, DbDeo to extract SQL statements and detect database schema smells. Practitioners may use these tools to identify quality issues impacting the maintainability of their software system.
- The experimental setup with all the required tools and scripts to apply deep learning on source code. The research community may build new ideas over it or replicate the experiment with the setup.



1.4 Research method

The journey of the research method starts from a literature review to understand the state of the art and to identify gaps in the current research and practice. The review reveals the potential opportunities for researchers in the field in general and provides a basis for defining the problem definition of the thesis.

In the design phase, we perform experimental design to address the problems that we identify. We aim to perform a large scale empirical study to detect smells in C# code and to carry out a study on architecture smells to identify their relationship with design smells. We conceptualize a feasibility study to detect smells using deep learning models and to apply transfer-learning in the smell detection context. We also design an exploratory study on configuration smells to first define configuration smells and support detection these smells in open-source repositories. Along similar lines, we put together a study for database schema smells; the study aims to reveal relationship between database quality and production code in the context of both open-source and close-source proprietary code. We identify relevant research questions to investigate for each experiment.

The implementation phase involves developing necessary tools and perform the experiments. We developed Designite to analyze C# code, Puppeteer for configuration smell detection, and DbDeo for database smell detection. We download open-source repositories from GitHub and perform our analysis on the them. Apart from quantitative analysis, we also perform qualitative analysis as well as developers' survey to strengthen the evaluation of the database analysis. We develop a set of tools and scripts to analyze and curate data from existing repositories. We put together the experimental setup to detect smells using deep learning and to establish feasibility of transfer-learning.

1.5 Thesis outline

This dissertation has six chapters. We outline the remaining five below.

Chapter 2 provides a holistic status quo of various dimensions associated with software smells. It presents the state-of-the-art in the current research concerning software smells, reveals deficiencies in present tools and techniques, and identifies research opportunities.


Chapter 3 illustrates the theoretical model and study design of our experiments along with specific research objectives for each experiment.


Chapter 4 provides implementation details of each experiment including tool support, and qualitative and quantitative analysis.


Chapter 5 presents results from our study and discusses the implications of the obtained results.





Chapter 6 presents conclusions of the thesis and chalks out some potential avenues for future extensions.


 **Convention used for highlighted boxes**
We use the following convention for the highlighted boxes in this thesis.


 **Goals**

 **Contributions**

 **Key results**

 **Implications**

 **Information or side note**

 **Opportunities**



Chapter 2

Related Work

The past is our teacher to build a better future. Well... only if we learn from it.

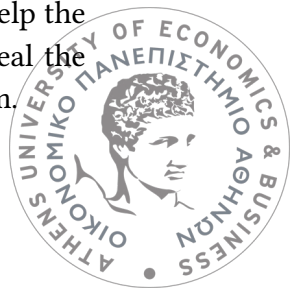
In this chapter, we present a comprehensive literature review of the domain of software smells. Section 2.1 introduces the term *code smells* and provides background of the related literature reviews. We discuss the method employed to perform the literature survey in Section 2.2 and define the research questions to explore. In Section 2.3, we present results and provide discussion for each addressed research question.

2.1 Introduction

Kent Beck coined the term “code smell” in the context of identifying quality issues in code that can be refactored to improve the maintainability of a software [Fow99]. He emphasized that the presence of excessive number of smells in a software system makes the software hard to maintain and evolve.

Since then, the smell metaphor has been extended to various related subdomains of software systems including testing [Deu01], database [Kar10], and configuration [SFS16]. Further, since the inception of the metaphor, the software engineering community has explored various associated dimensions that include proposing a catalog of smells, detecting smells using a variety of techniques, exploring the relationships among smells, and identifying the causes and effects of smells.

The large number of available resources poses a challenge, equally to researchers and practitioners, to comprehend the status quo of tools, methods, and techniques concerning software smells. Analyzing and synthesizing available information could not only help the software engineering community understand the existing knowledge, but also reveal the challenges that exist in the present set of methods and opportunities to address them.



There have been a few attempts to understand current practices and provide an overview of the existing knowledge about software smells. Singh *et al.* [SK17] present a systematic literature review on code smells and refactoring in object-oriented software systems by studying 238 primary studies. The survey focuses on the smell detection methods and tools as well as the techniques and tools used to refactor them. The authors divide smell detection methods based on the degree of automation employed in implementing smell detection methods.

Similarly, Zhang *et al.* [ZHB11] review studies from year 2000 to 2009 and draw a few observations about current research on smells. They reveal a large gap in existing smell literature – current studies have chosen a small number of smells for their study and some of the smells (such as message chains) are poorly explored by the community. Further, the study emphasizes that the impact of code smells is not well understood.

Various tools have been proposed to detect smells. Fernandes *et al.* [FOV⁺16] provide a comprehensive study containing a comparison of 84 smell detection tools. Similarly, Rasool *et al.* [RA15] also review existing code smell detection tools and reveal the challenges associated with them. A few studies [AD15, MT04] provide an extensive coverage to techniques available for refactoring smells.

In this chapter, we thoroughly explore the resources related to smells that were published between the years 1999 – 2016 and present the current knowledge in a synthesized and consolidated form. Additionally, our goal is to identify challenges in the present knowledge and find opportunities to overcome them.



Contributions

This survey makes the following contributions to the field.

- The study provides a holistic status quo of various dimensions associated with software smells. These dimensions include definition, classification, types, detection methods, as well as causes and impacts of smells.
- It presents the state-of-the-art in the current research, reveal deficiencies in present tools and techniques, and identifies research opportunities to advance the domain.

2.2 Method

In this section, we first present the objectives of this literature review and derived research questions. We illustrate the search protocol that we used to identify relevant studies. The search protocol includes not only the steps to collect the initial set of studies, but also inclusion and exclusion criteria that we apply on the initial set of studies to obtain a filtered set of primary studies.



2.2.1 Research objectives and questions



Research objectives

The goal of this study is to provide a consolidated yet extensive overview of software smells covering their definition, types, causes, detection methods, and impact on various aspects of software development.

In this study, we address the following research questions:

LR-RQ1 *What is the definition of a software smell?*

We aim to understand how the term “smell” is defined by various researchers. We infer basic defining characteristics and types of smells.

LR-RQ2 *How do smells get introduced in software systems?*

We explore the reasons that cause smells in software systems.

LR-RQ3 *How do smells affect the software development processes, artifacts, or people?*

We present the impact of smells on software systems. Specifically, we study impacts of smells on processes, artifacts, and people.

LR-RQ4 *How do smells get detected?*

We discuss the techniques employed by researchers to identify smells.

LR-RQ5 *What are the open research questions?*

We present the perceived deficiencies and the open research questions with respect to smells, their detection, and their interpretations.

The purpose of the prefix *LR* (i.e., *literature review*), which is used along with research questions labels, is to separate this set of research questions with other questions in the thesis.

2.2.2 Literature search protocol

The literature search protocol aims to identify primary studies which form the basis of the survey. Our search protocol has three phases:

1. We identify a list of relevant conferences and journals and manually search their proceedings.
2. We search seven well-known digital libraries.
3. We perform filtering and consolidation of the studies identified in the previous phases and prepare a single library of relevant studies.



2.2.2.1 Literature search – Phase 1

We identify a comprehensive list of conferences and journals based on papers published in these venues related to smells. We manually search the proceedings of the selected venues between the year 1999 and 2016. The start year has been selected as 1999 since the smell metaphor was introduced in 1999. During the manual search, the following set of terms were searched in the title of studies: *smell*, *antipattern*, *quality*, *maintainability*, *maintenance*, and *metric*. All the studies containing at least one of the search terms in their title were selected and recorded. Table 2.1 presents the selected conferences and journals along with their corresponding number of studies selected in Phase 1.

The domain of refactoring is closely related to that of software smells. However, given the vast knowledge present in the field of refactoring, it requires a separate study specifically for software refactoring. Therefore, we consider work concerning refactoring outside the scope of this study.

Table 2.1: Studies selected in the Phase 1

Venue	Type	#Studies
Automated Software Engineering	Conference	24
Empirical Software Engineering	Journal	61
Empirical Software Engineering and Measurement	Conference	68
European Conference on Object-Oriented Programming	Conference	2
Foundations of Software Engineering	Conference	19
IEEE Software	Journal	78
International Conference of Software Maintenance and Evolution	Conference	220
International Conference on Program Comprehension	Conference	38
International Conference on Software Engineering	Conference	85
Journal of Systems and Software	Journal	146
Mining Software Repositories	Conference	28
Software Analysis, Evolution, and Reengineering / European Conference on Software Maintenance and Reengineering	Conference	135
Source Code Analysis and Manipulation	Conference	22
Systems, Programming, Languages and Applications: Software for Humanity	Conference	8
Transactions on Software Engineering	Journal	83
Transactions on Software Engineering and Methodology	Journal	11
Total selected studies in Phase 1		1028



2.2.2.2 Literature search – Phase 2

In the second phase, we carried out search on seven well-known digital libraries. The terms used for the search are: *software smell*, *antipattern*, *software quality*, *maintainability*, *maintenance*, and *software metric*. We appended the term “software” to the search terms in order to obtain more relevant results. Additionally, we apply filters such as “computer science” and “software engineering” wherever it was possible and relevant to refine the search results. Table 2.2 shows the searched digital libraries and corresponding number of selected studies.

Table 2.2: Studies selected in the Phase 2

Digital Library	Number of studies
Google Scholar	196
SpringerLink	44
ACM Digital Library	108
ScienceDirect	40
Scopus	150
IEEE Xplore	151
Web of Science	58
Total selected studies in Phase 2	747

2.2.2.3 Literature search – Phase 3

In the third phase, we defined inclusion and exclusion criteria to filter out irrelevant studies and to prepare a consolidated library. The inclusion/exclusion criteria are listed below.

Inclusion criteria

- Studies that discuss smells in software development, present a catalog of one of the different types of software smells (such as code smells, test smells, and configuration smells), produce factors that cause smells, or explore their impact on any facet of software development (for instance, artifacts, people, or process).
- Studies introducing smell detection mechanisms or providing a comparison using any suitable technique.
- Resources revealing the deficiencies in the present set of methods, tools, and practices.

Exclusion criteria

- Studies focusing on external (in-use) software quality or not directly related with software smells.
- Studies that propose the refactoring of smells, or identifies refactoring opportunities.
- Articles comprising keynote, extended abstract, editorial, tutorial, poster, or panel discussion (due to insufficient details and small size).



- Studies whose full text is not available.

Each selected article from phase 1 or 2 went through a manual inspection of title, keywords, and abstract. The inspection applied the inclusion and exclusion criteria leading to inclusion or exclusion of the articles. We obtained 445 articles after completing the inspection and removing the duplicates. These articles are the primary studies that we studied in detail. We took notes while studying the selected articles. We then mark all the relevant articles for each research question and included them in the corresponding discussion.

We did not limit ourselves only to the primary studies. We included secondary sources of information and articles as and when we spotted them while studying primary studies. Therefore, although our primary studies belong to the period 1999 – 2016, due to the inclusion of the secondary studies, we refer studies in this survey that were published before or after the selected period. An interested reader may find the list of all the selected papers in each phase online [SS17].

After we completed the detailed study, we categorized the resources based on the dimensions of smells they belong to. Figure 2.1 provides an overview of the studied dimensions of software smells; a number in brackets shows the number of associated references.

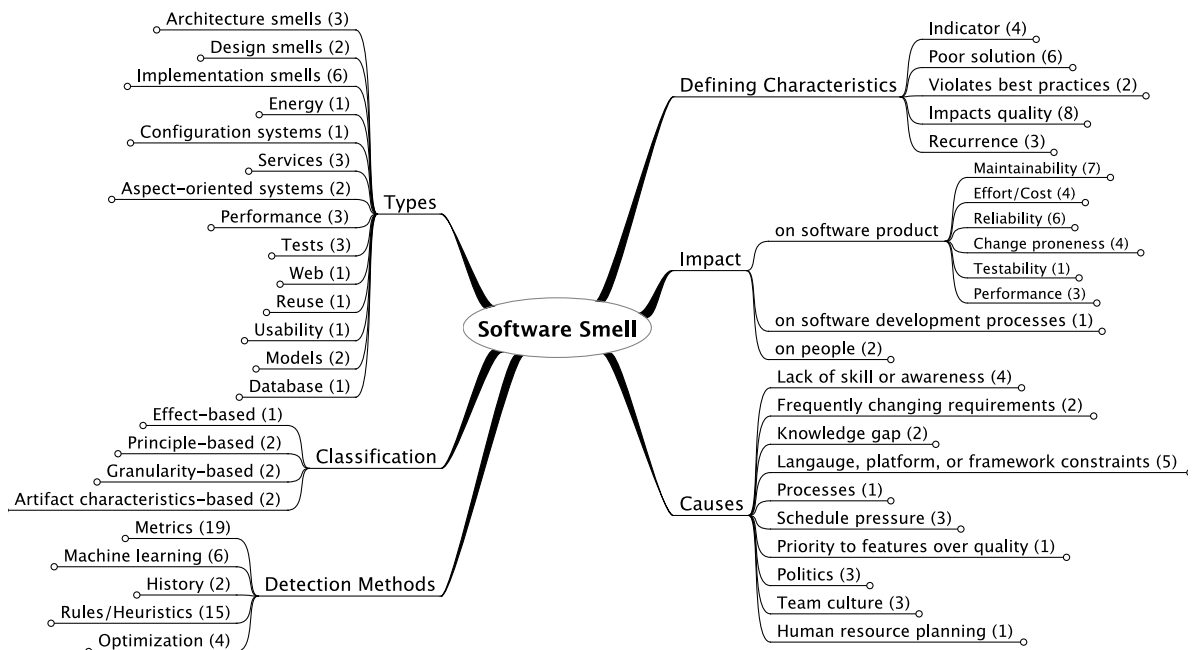


Figure 2.1: Overview of the study; a number in brackets shows the number of associated references

2.3 Results and Discussion

In this section, we present our synthesized observations corresponding to each research question addressed in this study.



2.3.1 LR-RQ1: What is the definition of a software smell?

We break down the question into the following sub-questions where each sub-question deals with precisely one specific aspect of software smells' definition.

LR-RQ1.1 *What are the defining characteristics of a software smell?*

LR-RQ1.2 *What are the types of smells?*

LR-RQ1.3 *How are the smells classified?*

LR-RQ1.4 *Are smells and antipatterns considered synonyms?*

2.3.1.1 LR-RQ1.1: What are the defining characteristics of a software smell?

Kent Beck coined the term “code smell” [Fow99] and defined it informally as “*certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring*”. Later, various researchers gave diverse definitions of software smells. A complete list of definitions of smells provided by various authors can be found in Appendix 6.3. Based on these, we synthesize the following five possible defining characteristics of a software smell.

- **Indicator:** Authors define smells as an indicator to or a symptom of a deeper design problem [MGDM10, Yam14, SSS14, dSS16].
- **Poor solution:** The literature describes smells as a suboptimal or poor solution [KVGS11, KEA16, FBA11, ADPAG13, VEM02, CDMT14].
- **Violates best practices:** According to authors such as Suryanarayana *et al.* [SSS14] and Sharma *et al.* [SFS16], smells violate recommended best practices of the domain.
- **Impacts quality:** Smells make it difficult for a software system to evolve and maintain [Yam14, KVGS11]. It is commonly agreed that smells impact the quality of the system [JGHK13, MG07, ADPAG13, GPDM09, SFS16, SSS14].
- **Recurrence:** Many authors define smells as recurring problems [MAB⁺12b, PZ12, KVGS11].

2.3.1.2 LR-RQ1.2: What are the types of smells?

Authors have explored smells in different domains and in different focus areas. Within software systems domain, authors have focused on specific aspects such as configuration systems, tests, and models. These explorations have resulted in various smell catalogs. Table 2.3 presents a summary of catalogs and corresponding references.

We have compiled an extensive catalog belonging to each focus area. Here, considering the space constraints, we provide a brief catalog of code smells in Table 2.4. We have selected the smells included in this table based on the popularity of the smells *i.e.*, based on

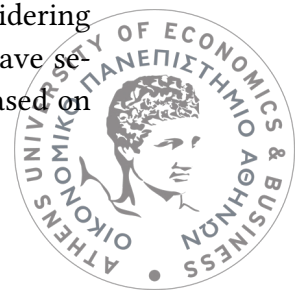


Table 2.3: Types of smells

Focus	References
Implementation	[Fow99], [ADPAG13], [BMMM98], [FM13], [AHTM11], [GKA ⁺ 16]
Design	[SSS14] [BGH ⁺ 08]
Architecture	[GPEM09], [BMMM98] [LK00]
Tests	[GvDS13], [HJE ⁺ 13] [Deu01]
Performance	[Smi00], [SA14], [WHTK14]
Configuration systems	[SFS16]
Database	[Kar10]
Aspect-oriented systems	[AFF14], [BGvS11]
Energy	[VAPM13]
Models	[EAM09], [DD16]
Services	[PDMG14], [KŽ07], [PM15]
Usability	[ACSS15]
Reuse	[Lon01]
Web	[NNN ⁺ 12]

the number of times the smell has been studied in the literature. The comprehensive and evolving taxonomy of software smells can be accessed online.¹

Table 2.4: Common code smells

Code Smell / References	Description
God class [Rie96]	The god class smell occurs when a huge class which is surrounded by many data classes acts as a controller (<i>i.e.</i> , takes most of the decisions and monopolises the functionality offered by the software). The class defines many data members and methods and exhibits low cohesion. Related smells: Insufficient modularization [SSS14], Blob [BMMM98], Brain class [VMDP14].
Feature envy [Fow99]	This smell occurs when a method seems more interested in a class other than the one it actually is in.
Shotgun surgery [Fow99]	This smell characterizes the situation when one kind of change leads to a lot of changes to multiple different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change.
Data class [Fow99]	This smell occurs when a class contains only fields and possibly getters/setters without any behavior (methods). Related smells: Broken modularization [SSS14].

¹<http://www.tusharma.in/smells>



Table 2.4: Common code smells

Code Smell / References	Description
Long method [Fow99]	This smell occurs when a method is too long to understand. Related smells: God method [Rie96], Brain method [VMDP14].
Functional decomposition [BMMM98]	This smell occurs when highly procedural and non-object-oriented code is written in an object-oriented language.
Refused bequest [Fow99]	This smell occurs when a subclass rejects some of the methods or properties offered by its superclass. Related smells: Rebellious hierarchy [SSS14]
Spaghetti code [BMMM98]	This smell refers to an unmaintainable, incomprehensible code without any structure. The smell does not exploit and prevents the use of object-orientation mechanisms and concepts.
Divergent change [Fow99]	Divergent change occurs when one class is commonly changed in different ways for different reasons. Related smells: Multifaceted abstraction [SSS14].
Long parameter list [Fow99]	This smell occurs when a method accepts a long list of parameters. Such lists are hard to understand and difficult to use.
Duplicate code [Fow99]	This smell occurs when same code structure is duplicated to multiple places within a software system. Related smells: Duplicate abstraction [SSS14], Unfactored hierarchy [SSS14], Cut and paste programming [BMMM98].
Cyclically-dependent modularization [SSS14]	This smell arises when two or more abstractions depend on each other directly or indirectly. Related smells: Dependency cycles [Mar01]
Deficient encapsulation [SSS14]	This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required. Related smells: Excessive global variables [FM13].
Lava flow [BMMM98]	This smell is characterized by a piece of code that nobody remembers the purpose and usage, and is largely not utilized. Related smells: Unutilized abstraction [SSS14].
Speculative generality [Fow99]	This smell occurs where an abstraction is created based on speculated requirements. It is often unnecessary that makes things difficult to understand and maintain. Related smells: Speculative hierarchy [SSS14]
Lazy class [Fow99]	This smell occurs where a class is not doing enough <i>i.e.</i> , it does not realize a concrete responsibility.



Table 2.4: Common code smells

Code Smell / References	Description
	Related smells: Unnecessary abstraction [SSS14].
Switch statement [Fow99]	This smell occurs when switch statements that switch on type codes are spread across the software system instead of exploiting polymorphism. Related smells: Unexploited encapsulation [SSS14], Missing hierarchy [SSS14].
Primitive obses- sion [Fow99]	This smell occurs when primitive data types are used where an abstraction encapsulating the primitives could serve better. Related smells: Missing abstraction [SSS14].
Swiss army knife [BMMM98]	This smell arises when the designer attempts to provide all possible uses of the class and ends up in an excessively complex class interface. Related smells: Multifaceted abstraction [SSS14].

We further elaborate this research question to focus on the related work corresponding to the investigations taken up in this thesis.

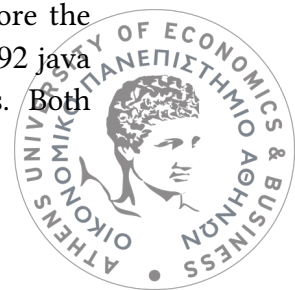
Quality Practices for Software Architecture

The topic of architecture smells and its impact on software development is a subject of interest for software engineering community for many years. Garcia *et al.* [GPem09] define an initial set of architecture smells and prepare a catalog of architecture smells with their mathematical definitions [Gar14]. These formal definitions are helpful in implementing smell detection tools for architecture smells. Brown *et al.* [BMMM98] also document a set of architecture smells in the enterprise settings.

Andrade *et al.* [dAAC14a] define a set of architecture smells with respect to Product Line Architecture (PLA). Mo *et al.* [MCKX15] identify a set of hotspot patterns, referring as recurring architecture problems based on a combination of historical and architectural information of software systems.

Yamashita *et al.* [YZFW15, YC13] empirically investigate the inter-smell relationships, termed as *collocated* and *coupled* that mainly comprise design smells and a few implementation smells. One of the insights from their work is that the explanatory power of code smell relationships need further investigations with complementary perspectives in order to be deemed useful. Inter-smell relationships have also been leveraged to find optimal smell-removing refactoring sequences [LMSN12].

Palomba *et al.* [Pal18] investigate the collocation (termed as co-occurrences) among 13 code (design and implementation) smells over multiple releases of 30 open source software systems. Similarly, Walter *et al.* [WFF18] conduct an experimental study to explore the collocation relationship among 14 code (design and implementation) smells across 92 java applications. They explore the effect of application domain on these relationships. Both



of these studies foresee the importance of smell collocations in identifying classes requiring high maintenance effort, and developing appropriate refactoring approaches and smell detection tools supporting collocation analysis.

It is believed that the software architecture is affected by the presence and criticality of code anomalies or smells. Macia *et al.* [MGP⁺12] present an empirical study on the relationship between code anomalies and architecture degradation. They conclude that 50% of the automatically detected code anomalies causes the architectural modularity problems.

Oizumi *et al.* [OGC⁺15, OGdSS⁺16] believe that the architecture problems are reflected in source code through groups of code smells and study the impact of a number of code smell agglomerations on architecture problems. Guimaraes *et al.* [GGC14, GVG⁺] conduct a controlled experiment utilizing architecture blueprints to prioritize various types of code smells based on their architecture relevance.

Martini *et al.* [MFBR18] conduct a case study based research on three architectural smells employing questionnaires, interviews, and code inspections on four industrial software projects. The main aim of this study is to identify and prioritize the architecture debt with the help of architecture smells. The findings of this study acknowledge the adverse effects of architecture smells, and emphasize on the unavailability of automatic smell detection tools. Le *et al.* [LLSM18] perform an empirical investigation on the nature and impact of six architecture smells that most frequently appeared in eight Apache Software Foundation open-source projects. They design detection algorithms for these smells and explored relationships between issues and the architecture smells under study. The outcome of this study states the negative impacts of architecture smells on maintenance effort in terms of increased number of implementation issues and code commits.

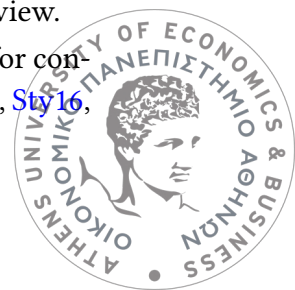
Quality Practices in System Configuration Management

In the landscape of system configuration management, empirical studies on configuration code written in languages such as Puppet [Pup18] and Chef [Che18] are scarce. Jiang *et al.* [JA15] study the co-evolution of Puppet and Chef configuration files with source, test, and build code. They analyze the software repositories of 256 OpenStack projects and distinguish files as infrastructure, which contain configuration code in Puppet or Chef language, production, build, and test. They find that configuration code comes in large files, changes more frequently, and presents tight coupling with test files.

Puppet Forge [Pup16b] — the repository of Puppet modules, provides an evaluation of configuration code quality through a quality score based on three aspects: code quality score provided by Puppet-Lint [Pup16c], compatibility with Puppet, and metadata quality. Metadata quality is subject to a set of guidelines that metadata files should adhere to.

Sonar-Puppet [Pup16a] is a SonarQube [Son16] plug-in that has numerous rules to detect quality violations in Puppet code; most of the rules applied by Sonar-Puppet are common with Puppet-Lint. Although the quality score provides useful and quick feedback to Puppet code authors, it is not near to a comprehensive code analysis from the IaC point of view.

In addition, various authors have published their ideas describing best practices for configuration code in the form of blog-posts, articles, and technical talks [Lar16a, Lar16b, Style,



Lar16c].

Quality Practices in Database Applications

There is scant research that explores the quality characteristics of database code. Karwin [Kar10] presents a comprehensive catalog of database antipatterns drawn from industry experience. He organizes antipatterns in four categories: logical database design, physical database design, query, and application development antipatterns.

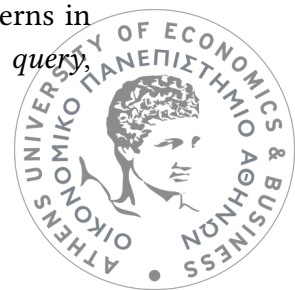
Authors have attempted studies to explore the quality aspect of database code. Brink *et al.* [BLV07] discuss the challenges in extracting SQL statements from the host source code and present a method to extract and distil SQL statements. The study provides a set of basic metrics concerning databases such as number of tables and nested queries. Chen [Che15] proposes strategies for reducing the impedance mismatch between the relational and object-oriented models in order to improve database performance and integrity.

The knowledge and experience accumulated in popular question and answer sites can be leveraged to help developers avoid smells in SQL queries. Nagy *et al.* [NC15] mine Stack Overflow questions that are relevant to SQL queries. The study extracts SQL error patterns as a first step towards a recommendation system that aids developers to construct correct queries. Eessaar [EV15] also discusses a few heuristics that can be employed to detect some of the database smells outlined by Karwin [Kar10]. Many authors have explored object-relational mapping in the context of their implications on application design [TGPM17] and performance [CSJ⁺14].

2.3.1.3 LR-RQ1.3: How are the smells classified?

An appropriate classification is required to better comprehend a long list of smells based on their characteristics. We collected, categorized, inferred, and synthesized the following set of meta-classification of software smells.

- **Effect-based smell classification:** Mäntylä *et al.* [MVL03] classified smells based on their effects on software development activities. The categories provided by the classification include *bloaters*, *couplers*, and *change preventers*.
- **Principle-based smell classification:** Samarthayam *et al.* [SSSG13] and Suryanarayana *et al.* [SSS14] classified design smells based on the primary object-oriented design principles that the smells violate. The principle-based classification divided the smells in four categories namely: *abstraction*, *modularization*, *encapsulation*, and *hierarchy* smells.
- **Artifact characteristics-based smell classification:** Wake [Wak03] proposed a smell classification based on characteristics of the types (i.e., classes or interfaces). Categories such as *data*, *interfaces*, *responsibility*, and *unnecessary complexity* are included in his classification. Similarly, Karwin [Kar10] classified SQL antipatterns in the following categories — *logical database design*, *physical database design*, *query*, and *application development* antipatterns.



- **Granularity-based smell classification:** Moha *et al.* [MGDM10] classified smells using a two-level classification. At first, a smell is classified in either *inter-class* and *intra-class* category. The second level of classification assigns non-orthogonal categories *i.e.*, *structural*, *lexical*, and *measurable* to the smells. Similarly, Brown *et al.* [BMMM98] discussed antipatterns classified in three major categories — *software development*, *software architecture*, and *software project management* antipatterns.



Desirable properties of a classification

Kenneth Bailey [Bai94] discusses a few desirable properties of a classification. By applying them in the context of our study, we propose that an ideal classification of smells must exhibit the following properties.

- **Exhaustive:** classify all the considered smells,
- **Simple:** classify smells within the scope and granularity effortlessly,
- **Consistent:** produce a consistent classification even if it carried out by different people, and
- **Coherent:** produce clearly distinguishable categories without overlaps.

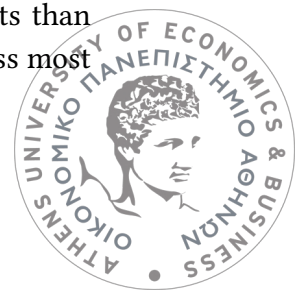
We encourage authors that propose a classifications, in general within software engineering context or specifically for smells, adhere to the above-mentioned properties.

2.3.1.4 LR-RQ1.4: Are smells and antipatterns considered synonyms?

Software engineering researchers and practitioners often use the terms “antipattern” and “smell” interchangeably. Specifically, authors such as Palma *et al.* [PMG13], Palomba *et al.* [PNT⁺15], and Linares *et al.* [LVKM⁺14] use both the terms as synonyms. For instance, Linares *et al.* [LVKM⁺14] assert this notion explicitly — “...we use the word “smells” to refer to both code smells and antipatterns, ...”

Some authors treat antipatterns and smells as quality defects at different granularity. For example, Moha *et al.* [MG07, MGDM10] defined design defects as antipatterns at design granularity and as smells at implementation granularity.

Andrew Koenig [Koe95] coined the term “antipatterns” in 1995 and defined it as follows: “An antipattern is just like pattern, except that instead of solution it gives something that looks superficially like a solution, but isn’t one”. Hallal *et al.* [HAT⁺04] also describe antipatterns in this vein — “something that looks like a good idea, but which backfires badly when applied”. Based on Koenig’s definition, our following interpretation makes antipatterns fundamentally different from smells — *antipatterns get chosen but smells occur, mostly inadvertently*. An antipattern is chosen in the assumption that the usage will bring more benefits than liabilities whereas smells get introduced due to the lack of knowledge and awareness most of the times.



Brown *et al.* [BMMM98] specify one key characteristic of antipatterns as one “...that generates decidedly negative consequences.” This characteristic makes antipatterns significantly different from smells — a smell is considered as an *indicator* (refer Section 2.3.1.1) of a problem (rather than the problem itself) whereas antipatterns bring decidedly negative consequences.

An antipattern may lead to smells. For instance, a variant of Singleton introduces subtype knowledge in a base class leading to cyclic hierarchy [SSS14] smell in the code [FDW⁺16]. Further, the presence of smells may indicate that a certain practice is an antipattern rather than a recommended practice in a given context. For example, the Singleton pattern makes an abstraction difficult to test and hence introduces test smells; the presence of test smells helps us identify that the employed pattern is deteriorating the quality more than helping us solving a design problem.



Implications

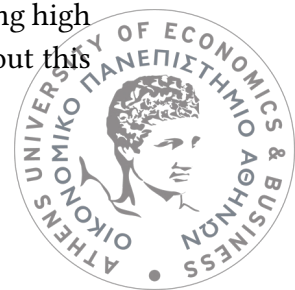
We can draw the following implications from the above-discussed research question.

- We found that smells may occur in various stages of software development and impair many dimensions of software quality of different artifact types. This implies that software developers should adopt practices to avoid smells at different granularities, artifacts, and quality dimensions at all stages of software development.
- We identified the core characteristics of software smells. This can help the research community to identify smells even when they are not tagged as smells. For example, it is a recommended practice to avoid accessing external dependencies, such as a database, in a unit test [Bec02]. A code fragment violating this recommended practice shows presence of code smells as the fragment exhibits properties *violates best practices* and *impacts quality* (maintainability and performance) of a smell. Therefore, such a violation of the recommended practice could be identified as a test smell despite not being referred to as a smell.
- We elaborated on the distinction between antipatterns and smells. This distinction can be further explored in future research on these topics.

2.3.2 LR-RQ2: How do smells get introduced in software systems?

Authors have explored factors that introduce smells in software systems. We classify such causal factors into the following consolidated list.

- **C1: Lack of skill or awareness:** A major reason that cause smells in software systems is poor technical skills of developers and lack of awareness towards writing high quality code. Many authors [SSS14, MBC14, CMRP16, TAV13] have pointed out this cause in their studies.



- **C2: Frequently changing requirements:** Certain design decisions are made to fulfil the requirements at hand; however, frequent changes in requirements impair the effective decision making and introduce smells [MBC14, LR15].
- **C3: Language, platform, or technology constraints:** The current literature [SJW08, MBC14, KHRS12, LR15, CMRP16] shows that the chosen technology influences design decisions and could be another reason that leads to smells.
- **C4: Knowledge gap:** Missing or complex documentation introduces a knowledge gap which in turn could lead to smells in a software [LR15, MBC14].
- **C5: Processes:** The adopted processes may help avoid smells to occur or remain in a software system. Therefore, an ineffective or a missing set of processes could also become a cause for software smells [TAV13, SSS15].
- **C6: Schedule pressure:** Developers adopt a quick fix rather than an appropriate solution in the scarcity of time. These quick fixes are a source of smells in software systems [LR15, MBC14, SSS14].
- **C7: Priority to features over quality:** Managers consistently pressurise the development teams to deliver new features quickly and ignore the quality of the system [MBC14].
- **C8: Politics:** Organizational politics for control, position, and power influence the software quality [CMRP16, LR15, SM06].
- **C9: Team culture:** Many authors [AGJ08, CMRP16, TAV13] have recognized the practices and the culture prevailed within a team or an organization as a cause of software smells.
- **C10: Poor human resource planning:** Poor planning of human resources required for a software project may force the present development team to adopt quick fixes to meet the deadlines [LR15].

A cause-based classification can help us understand the categories of factors that causes smells. We propose an alternative to cause-based classification in the form of *actor-based classification*. The actor-based classification assigns the responsibility of the causes to specific actor(s). The identified actors should either correct smells in the current project or learn from the experience so as to avoid these smells in the future. For example, in the current context, we consider three actors — *manager* (representing individuals in the management hierarchy), *technical lead* (the person leading the technical efforts of a software development team), and a software *developer*. Table 2.5 presents the classification of causes following the actor-based classification scheme. Such a classification can help us in identifying the actionable tasks. For example, if the skill or awareness of software developers is lacking, the

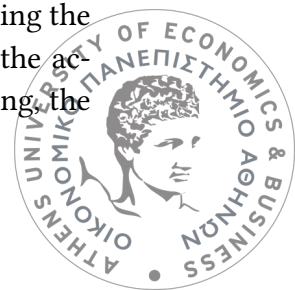


Table 2.5: Actor-based Classification of Smells Causes

Actor\Causes	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Manager						✓	✓	✓	✓	✓
Technical lead	✓	✓	✓	✓	✓				✓	
Developer	✓			✓					✓	

actor-based classification suggests that developers as well as their technical-leads are responsible to take a corrective action. Similarly, if appropriate processes are not in-place, it is the responsibility of the technical-lead to deploy them.

The above discussed roles and responsibility assignment is an indicative example. The classification has to be adapted based on the team dynamics and the context. For instance, the roles could differ in software development teams that follow different development methods (e.g. agile, traditional waterfall, and hybrid). Furthermore, some development teams are mature to take collective decisions whereas some teams have roles such as scrum master to take decisions that impact the whole team.



Implications

The above exploration consolidates factors reported in the literature that cause smells. It would be interesting to observe their comparative degree of impact on software smells. Further, we propose a classification that identifies the actors responsible to correct or avoid the causes of specific smells. This explicit identification of responsible actors is actionable; software development teams can improve code quality by making the actors accountable and working with them to correct the underlying factors that lead to specific smells.

2.3.3 LR-RQ3: How do smells affect the software development processes, artifacts, or people?

Smells impact not only software product but also the processes and people working on the product. Table 2.6 summarizes the impact of smells on software product, process, and people.

Smells have multi-fold impact on the artifacts produced in the software development process and associated quality. Specifically, smells impact maintainability, reliability, testability, performance, and change-proneness of the software. Further, smells also increase effort (and hence cost) required to produce a software.

Presence of excessive amount of smells in a product may influence the outcome of a process; for instance, a high number of smells in a piece of code may lead to pull request rejection [SVT16].

A high number of smells (and hence high technical debt) negatively impact the morale and motivation of the development team and may lead to high attrition [TAV13, SSS14].



Table 2.6: Impact of Smells

Entity	Attribute	References
Software product	Maintainability	[BQO ⁺ 12], [PR11], [MY12], [YM13a], [Yam14], [YM13b], [SYKG16]
	Effort/Cost	[SYA ⁺ 13], [SZV ⁺ 13], [SDPAG13], [MS16]
	Reliability	[JGHK13], [HZBS14], [ZSSS11], [BQO ⁺ 14], [MNK ⁺ 02], [KDPGA12]
	Change proneness	[OCBZ09], [KDPGA12], [ZSSS11], [KDPG09]
	Testability	[SDPAG13]
	Performance	[CSJ ⁺ 14], [HMR16], [SA14]
Software development Processes		[SVT16]
People	Morale and motivation	[TAV13], [SSS14]
	Productivity	[TAV13]



Implications

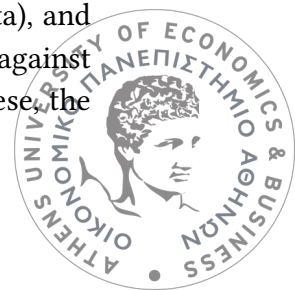
The above exploration reveals that impact of smells on certain aspects has not been studied in detail. For example, the impact of smells on testability of a software system and productivity of a software development team have been studied only by one study each. Further research in this area can quantify the degree of the smells' impact on diverse product and process quality aspects along with the corresponding implications.

2.3.4 LR-RQ4: How do smells get detected?

A large body of knowledge exists to detect software smells. Smells have been detected in many studies by employing various techniques. We classify the smell detection strategies in five broad categories; we describe these categories below. Figure 2.2 shows an abstract layered process flow that we have synthesized by analyzing existing approaches to detect smells using the five categories of smell detection.

1. **metric-based smell detection:** A typical metric-based smell detection method takes source code as the input, prepares a source code model, such as an AST (Abstract Syntax Tree), (step 1.1 in the figure 2.2) typically by using a third-party library, detects a set of source code metrics (step 1.2) that capture the characteristics of a set of smells, and detects smells (step 1.3) by applying a suitable threshold [Mar05].

For example, an instance of the god class smell can be detected using the following set of metrics: wmc (Weighted Methods per Class), ATFD (Access To Foreign Data), and tcc (Tight Class Cohesion) [Mar04, VMDP14]. These metrics are compared against pre-defined thresholds and combined using logical operators. Apart from these, the



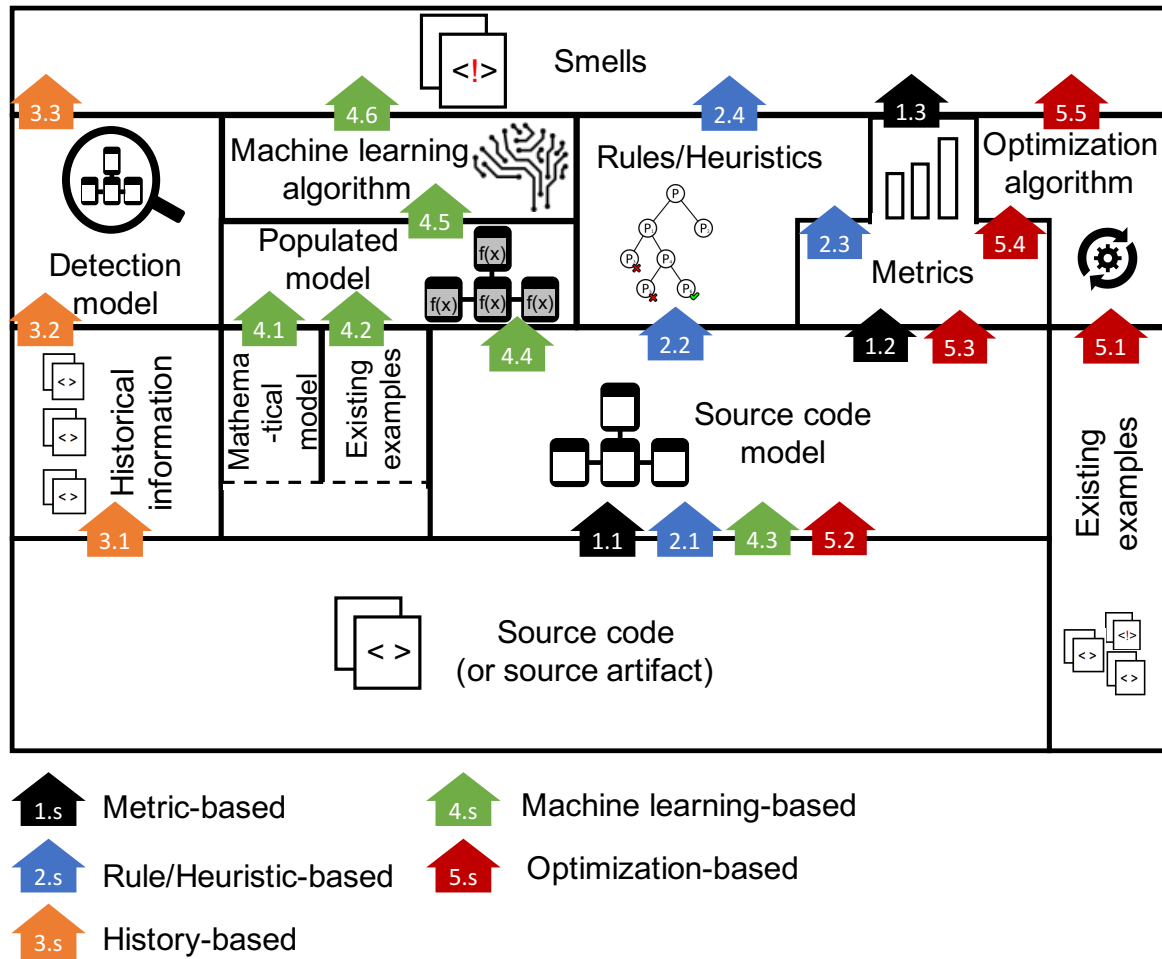
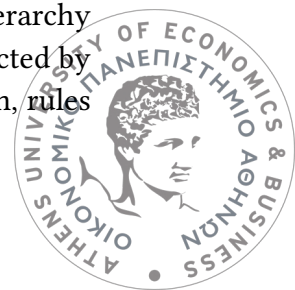


Figure 2.2: A layered overview of smell detection methods. Each detection method starts from the code (or other source artifact) and goes through various steps to detect smells. The direction of the arrows shows the flow direction and annotations on the arrows show the detection method (first part) and the step number (second part).

community frequently uses other metrics such as NOC (Number of Children), NOM (Number of Methods), CBO (Coupling Between Objects), RFC (Response For Class), and LCOM (Lack of Cohesion of Methods) [CK94] to detect other smells.

- Rules/Heuristic-based smell detection:** Smell detection methods that define rules or heuristics [MGDM10] (step 2.2 in the figure 2.2) typically takes source code model (step 2.1) and sometimes additional software metrics (step 2.3) as inputs. They detect a set of smells when the defined rules/heuristics get satisfied.

There are many smells that cannot be detected by the currently available metrics alone. For example, we cannot detect rebellious hierarchy, missing abstraction, cyclic hierarchy, and empty catch block smells using commonly used metrics. In such cases, rules or heuristics can be used to detect smells. For example, the cyclic hierarchy [SSS14] smell (when the supertype has knowledge about its subtypes) is detected by defining a rule that checks whether a class is referring to its subclasses. Often, rules



or heuristics are combined with metrics to detect smells.

3. **History-based smell detection:** Some authors have detected smells by using source code evolution information [PBDP⁺15]. Such methods extract structural information of the code and how it has changed over a period of time (step 3.1 in the figure 2.2). This information is used by a detection model (step 3.2) to infer smells in the code. For example, by applying association rule mining on a set of methods that have been changed and committed often to the version control system together, divergent change smell can be detected [PBDP⁺15].
4. **Machine learning-based smell detection:** In the recent times, *machine learning-based smell detection* methods have attracted software engineering researchers. Various machine learning methods such as Support Vector Machines [MAB⁺12b], and Bayesian Belief Networks [KVG09] have been used to detect smells. A typical machine learning method starts with a mathematical model representing the smell detection problem (step 4.1 in the figure 2.2). Existing examples (step 4.2) and source code model (step 4.3 and 4.4) could be used to instantiate a concrete populated model. The method results in a set of detected smells by applying a chosen machine learning algorithm (step 4.5) on the populated model. We elaborate on the machine learning-based methods in the next sub-section in greater detail.
5. **Optimization-based smell detection:** Approaches in this category apply optimization algorithms such as genetic algorithms [OKKI15] to detect smells. Such methods apply an optimization algorithm on computed software metrics (step 5.4 in the figure 2.2) and, in some cases, existing examples (step 5.1) of smells to detect new smells in the source code.

Among the surveyed papers, we selected all the papers that employ a smell detection mechanism. We classify these attempts based on the employed smell detection method. Table 2.7 shows existing attempts to identify smells using one of the smell detection methods. The table also shows number of smells detected by each of the method and target language/artifact.

Table 2.7: Smell Detection Methods and Corresponding References

Smell detection method	Reference	#Smells	Languages/ Artifacts
	[DPXT13]	1	Java
	[Mar05]	10	Java, C++
	[Mun05]	2	Java
	[SLT06]	5	Java
	[VRDBDR07]	2	Java
	[MHB10]	1	Java
	[OKAG10]	1	Java



Table 2.7: Smell Detection Methods and Corresponding References

Smell method	detection	Reference	#Smells	Languages/ Artifacts
		[MGvS10]	11	Java
		[FBA11]	5	UML Diagrams
		[BGvS11]	7	Aspects-oriented systems
		[SA13]	3	Java
		[FM13]	13	JavaScript
		[VMDP14]	10	Java
		[PPF ⁺ 14]	2	Java
		[APFC15]	3	NA
		[FSMS15]	1	C
		[Non15]	1	Java
		[VVDP ⁺ 16]	10	JavaScript
		[OCBZ09]	2	Java
Machine learning-based		[KVGs09]	1	Java
		[BBEAM10]	1	Java
		[KVGs11]	3	Java
		[MAB ⁺ 12b]	4	Java
		[CMC15]	1	Java
		[MKMD16]	NA	Java
History-based		[FS15]	3	Java
		[PBDP ⁺ 15]	5	Java
Rule/Heuristics-based		[Ram10]	5	C
		[EAM09]	8	Use-case Model
		[AHTM11]	8	C++
		[FTC07]	1	Java
		[TCC08]	1	Java
		[TC11]	1	Java
		[CMRT10]	4	UML Models
		[ABT15]	1	UML Models
		[PPDL ⁺ 16]	5	Java
		[LCCY13]	1	Java
		[PDMG14]	8	REST APIs
		[MGDM10]	4	Java
		[TK11]	6	Palladio Component Model
		[ADPAG13]	17	Java
		[SMT16]	30	C#
Optimization-based		[KKS ⁺ 14]	8	Java



Table 2.7: Smell Detection Methods and Corresponding References

Smell method	detection	Reference	#Smells	Languages/ Artifacts
		[SKBD14]	7	Java
		[GEBK15]	3	Java
		[OKKI15]	5	XML (WSDL)

Each detection method comes with a set of strengths and weaknesses. metric-based smell detection is convenient and relatively easy to implement; however, as discussed before, one cannot detect many smells using only commonly known metrics. Another important criticism of metric-based methods is their dependence on choosing an appropriate set of thresholds, which is a non-trivial challenge. Rule/Heuristic-based detection methods expand the horizon of metric-based detection by strengthening them with the power of heuristics defined on source code entities. Therefore, rule/heuristic-based methods combined with metrics offer detection mechanisms that can reveal a high proportion of known smells. History-based methods have a limited applicability because only a few smells are associated with evolutionary changes. Therefore, a source code entity (such as a method or a class) that has not necessarily evolved in a certain way to suffer from a smell cannot be detected by history-based methods. Machine learning approaches depend heavily on training data and the lack of such training datasets is a concern [KVG09]. Also, it is still unknown whether machine learning-based detection algorithms can scale to the large number of known smells. Further, optimization-based smell detection methods depend on metric data and corresponding thresholds. This fact makes them suffer from limitations similar to metric-based methods.

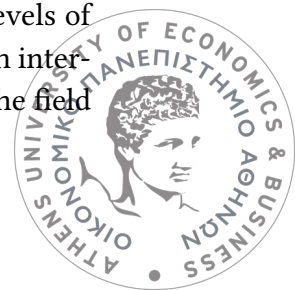
2.3.4.1 Machine learning techniques on source code

In this section, we present an introduction to deep learning and machine learning techniques applied on source code including code smells detection.

Introduction to deep learning

Machine learning is a subfield of artificial intelligence that *trains* solutions to problems rather than modeling them through hard-coded rules. In this approach, the rules that solve a problem are not set a-priori; rather, they are inferred in a data-driven manner. In supervised learning, a model is trained by being exposed to examples of instances of the problem along with their expected answers and statistical regularities are drawn. The representations that are learned from the data can in turn be applied and generalized to new, unseen data in a similar context.

Deep learning is a subfield of machine learning that allows computational models composed of multiple processing layers to learn representations of data with multiple levels of abstraction [LBH15, GBCB16]. Even though the idea of layered neural networks with internal “hidden” units was already introduced in the 80s [RHW86], a breakthrough in the field



came in 2006 by Hinton *et al.* [HOT06] who introduced the idea of learning a hierarchy of features one level at a time. Ever since, and particularly during the course of the last decade, the field has taken off due to the advances in hardware, the release of benchmark datasets [DDS⁺09, KH09, LCB10], and a growing research focus on optimization methods [Mar10, KB14]. Although deep learning architectures often consist of tens or hundreds of successive layers, much shallower architectures may also fall under the category of deep learning, as long as at least one hidden layer exists between the input and the output layer.

Deep learning architectures are being used extensively for addressing a multitude of detection, classification, and prediction problems. Architectures involving layers of CNNs are inspired by the hierarchical organization of the visual cortex in animals, which consists of alternating layers of simple and complex cells [FVE91, HW62]. CNNs have been proven particularly effective for problems of optical recognition and are widely used for image classification and detection [KSH12, SLJ⁺15, LBBH98], segmentation of regions of interest in biological images [KBF16], and face recognition [LGTB97, PVZ⁺15]. Besides recognition of directly interpretable visual features of an image, CNNs have also been used for pattern recognition in signal spectrograms, with applications in speech recognition [SKS⁺15]. In these applications the input data are given in the form of matrices (2D arrays) for representing the 2D grid layout of pixels in an image. 1D representations of data have been used for applying 1D convolutions in sequential data such as textual patterns [JZ15] or temporal event patterns [LYC17, AAK⁺17]. However, when it comes to sequential data, Recurrent Neural Networks (RNNs) [RHW86] have been proven superior due to their capability to dynamically “memorize” information provided in previous states and incorporate it to a current state. Long Short Term Memory (LSTM) networks are a special kind of RNN that can connect information spanning long-term intervals, thus capturing long-term dependencies. LSTMs have been found to perform reasonably well on various data sets within the context of representative applications that exhibit sequential patterns, such as speech recognition and music modeling [GSK⁺17, GJM13]. In addition, they have been established as state-of-the-art networks for a variety of natural language processing tasks; indicative applications include natural language generation [WGM⁺15], sentiment classification [WHZ⁺16, BPD17] and neural machine translation [CvMG⁺14], among others.

Machine learning techniques on source code

The emergence of online open-source repository hosting platforms such as GitHub in recent years has led to an explosion on the volumes of openly available source code along with metadata related to software development activities; this bulk of data is often referred to as “Big Code” [ABDS18]. As an effect, software maintenance activities have started leveraging the wealth of openly available data, the availability of computational resources, and the recent advances in machine learning research. In this context, statistical regularities observed in source code have revealed the repetitive and predictable nature of programming languages, which has been compared to that of natural languages [HBS⁺12, Ern17]. To this end, problems of automation in natural language processing, such as identification of semantic similarity between texts, translation, text summarisation, word prediction



and language generation have been examined in parallel with the automation of software development tasks. Relevant problems in software development include clone detection [WTV¹⁶, WL¹⁷], de-obfuscation [VCD¹⁷], language migration [NNN¹³], source code summarisation [IKCZ¹⁶], auto-correction [PNSLB¹⁶, GPKS¹⁷], auto-completion [FGL¹²], generation [OFN⁺¹⁵, LBG⁺¹⁶, YN¹⁷], and comprehension [APG¹⁷].

On a par with equivalent problems in natural language processing, the methods employed to address these software engineering problems have switched from traditional rule-based and probabilistic n-gram models to deep learning methods. The majority of the proposed deep learning solutions rely on the use of RNNs which provide sophisticated mechanisms for capturing long term dependencies in sequential data, and specifically LSTMs [HS⁹⁷] that have demonstrated particularly effective performance on natural language processing problems.

Alternative approaches to mining source code have employed CNNs in order to learn features from source code. Li *et al.* [LHZL¹⁷] have used a single-dimension CNNs to learn semantic and structural features of programs by working at the AST level of granularity and combining the learned features with traditional hand-crafted features to predict software defects. This method however incorporates hand-crafted features in the learning process and is not proven to yield transferable results. Similarly, a one-dimensional CNN-based architecture has been used by Allamanis *et al.* [APS¹⁶] in order to detect patterns in source code and identify “interesting” locations where attention should be focused. The objective of the study is to predict short and descriptive names of source code snippets (e.g., a method body) given solely its tokens. CNNs have also been used by Huo *et al.* [HLZ¹⁶] in order to address the problem of bug localization. This approach leverages both the lexical information expressed in the natural language of a bug report and the structural information of source code in order to learn unified features. A more coarse-grain approach that also employs CNNs has been proposed in the context of program comprehension [OAH⁺¹⁸] where the authors use imagery rather than text in order to discriminate between scripts written in two programming languages, namely Java and Python.

Code smells detection using machine learning techniques

Foutse *et al.* [KVGS⁰⁹, KVGS¹¹] use a Bayesian approach for the detection of smells. Their study forms a Bayesian graph using a set of metrics and determines the probability whether a class belongs to a smell or not. Similarly, Abdou *et al.* [MAB⁺^{12b}, MAB⁺^{12a}] employ support vector machine-based classifiers trained using a set of 60 object-oriented metrics for each class to detect design smells (*blob*, *feature concentration*, *spaghetti code*, and *swiss army knife*). Furthermore, Sérgio *et al.* [BBEAM¹⁰] detect *long method* smell instances by employing binary logistic regression. They use commonly used method metrics, such as Method Lines of Code (MLOC) and cyclomatic complexity as regressors. Bardez *et al.* [BKG¹⁹] presents an ensemble method that combine outcome of multiple tools to detect *god class* and *feature envy* smells. They identify a set of key metrics for each smell and feed them to a CNN-based architecture. Fontana *et al.* [FPRZ¹⁶] compare performance of various



machine learning algorithms in detecting *data class*, *god class*, *feature envy*, and *long method*.

However, machine learning techniques to detect smells are considered far from mature. In a recent study, Di Nucci *et al.* [NPT⁺18] note that the problem of detecting smells still requires extensive research to attain a maturity that would produce results of practical use. In addition, machine learning techniques (such as Bayesian networks, support vector machines, and logistic regression) that have been applied so far require considerable pre-processing to generate features for the source code, a substantial effort that hinders their adoption in practice. Traditionally, researchers use machine-learning methods that require extracting feature-sets from source code. Typically, code metrics are used as the feature set for smell detection purposes. We perceive two shortcomings in such usage of machine-learning methods for detecting smells. First, we need an external tool to compute metrics for the target programming language on which we would like to apply the machine learning model. Those that have a metrics computation tool may deduce many smells directly by combining these metrics [Mar04, SS18] and thus applying a machine-learning method is redundant. Second and more importantly, we are limiting the machine learning algorithm to use only the metrics that we are computing and feeding as feature-set. Therefore, the machine learning algorithm cannot observe any pattern that is not captured by the provided set of metrics.



Implications

We identify five categories of smell detection mechanisms. An implication of the categorization for the research community is the positioning of new smell detection methods; the authors can classify their new methods as one of these categories or propose a new smell detection method category.

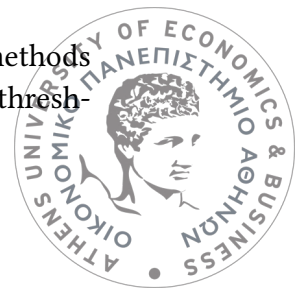
Among the five types of smell detection methods, metric-based tools are the most popular and relatively easier to develop. On the other hand, researchers are attracted towards machine learning-based methods to overcome the shortcomings of other smell detection methods such as the dependence on choosing appropriate threshold values for metrics. However, many challenges remain in using machine learning techniques. The availability of a standard training dataset and advancements in employing deep learning models would encourage researchers to develop better smell detection tools using machine learning approaches.

2.3.5 LR-RQ5: What are the open research questions?

Despite the availability of huge amount of literature on smells and associated aspects, we perceive many opportunities to expand the domain knowledge.

1 False-positives and lack of context: Results produced by the present set of smell detection tools are prone to false-positive instances [FDW⁺16, KVG11].

- The major reason of the false-positive proneness of the smell detection methods is that metrics and rule-based methods depend heavily on the metrics thresh-



olds. The software engineering community has identified threshold selection as a challenge [KKS⁺14], [FBA11]. There have been many attempts to identify optimal thresholds [FFZY15, LLNL16, FBB⁺12]; however, the proneness to false-positives cannot be eliminated in metrics and rule-based methods since one set of thresholds (or a method to derive thresholds) do not apply in another context.

- Many authors have asserted that smell detection is a subjective process [ML06, PBP⁺14, MHB08]. As Gil *et al.* [GL16] say – “Bluntly, the code metric values, when inspected out of context, mean nothing.” Similarly, Fontana *et al.* [FDW⁺16] list a set of commonly detected smells that solve a specific design problem in the real-world.

We suggest that the smells identified using tools must go through an expert-based scrutiny to finally tag them as quality problems. Essentially, the present set of smell detection methods are not designed to take *context* into account. One potential reason is that it is not easy to define, specify, and capture context. This presents an interesting yet challenging opportunity to significantly improve the *relevance* of detected smells.

- Another interesting concern related to smells in the context of false-positives is that smells are *indicative* by definition and thus it is unfair to tag smells as false-positive based on the context. As shown in Figure 2.3, a recorded smell could be a false-positive instance (and thus not an actual smell) when it does not fulfill the criteria of a smell by the definition of a smell. When the recorded smell is not a false-positive instance, it could either be a smell which is not a quality problem considering the context of the detected smell or it could be a definite quality problem contributing to technical debt. **This brings up the interesting insight that researchers and practitioners need to perceive smells (as indicators) differently from definite quality problems.**

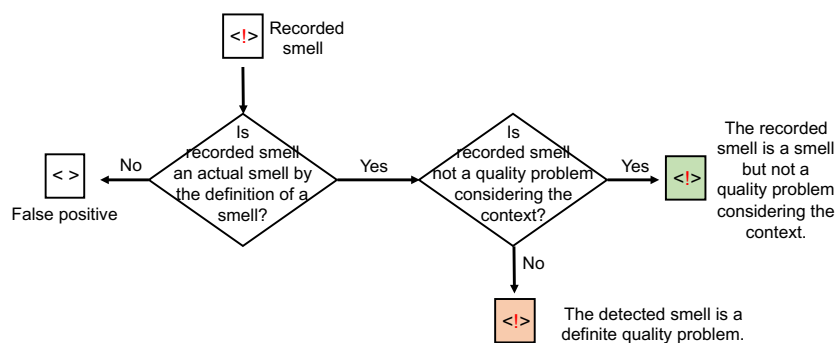
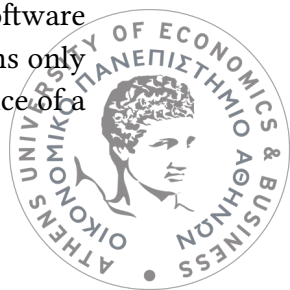


Figure 2.3: A recorded smell could be a false-positive instance, a smell that is not a quality problem, or a definite quality problem.

For example, consider a tool reports an instance of data class smell in a software system. As explained in Table 2.4, this smell occurs when a class contains only data fields without any methods. A common practice is to tag the instance of a



data class as a false-positive when it is serving a specific purpose in that context [FDW⁺16]. However, we argue that rather than tagging the instance as a false-positive (based on the context), we define smells as being separate from the definite quality problems. A fowl smell in a restaurant may indicate something is rotten, but can also accompany the serving of a strongly smelling cheese.

In a manual inspection, if we find that the class has one method apart from data fields then the reported smell is a false-positive instance since it does not fulfill the condition of a data class smell. On the other hand, if the class only contains data fields without any method definition, it is a smell. As a developer, if one considers the context of the class and infers that the class is being used, for instance, as a DTO (Data Transfer Object) [Fow02] the smell is not a quality problem because it is the result of a conscious design decision. However, if the above case does not apply and the developer is using another class (typically a *manager* or a *controller* class) to access and manipulate the data members of the data class, the identified smell is a definite quality problem.

2 Limited detection support for known smells: Table 2.8 shows all the smell detection tools selected in this study and their corresponding supported smells. It is evident that most of the existing tools support detection of a significantly smaller subset of known smells. Researchers [PDLBO14, RA15, SMT16] have identified the limited support present for identifying smells in the existing literature. The deficiency poses a serious threat to empirical studies that base their research on a severely low number of smells.



Table 2.8: Smell Detection Methods and supported smells

References	Detection Method	God class	Feature envy	Shotgun surgery	Data class	Long method	Functional decomposition	Refused bequest	Spaghetti code	Divergent Change	Long Parameter List	Other smells	Total smells
[Mar05]	metric-based	✓	✓	✓	✓	✓		✓				4	10
[Mun05]	metric-based											2	2
[SLT06]	metric-based	✓	✓	✓		✓		✓				0	5
[VRDBDR07]	metric-based											2	2
[FTC07]	Rule/Heuristic-based		✓									0	1
[TCC08]	Rule/Heuristic-based											1	1
[KVGS09]	Machine learning-based	✓										0	1
[EAM09]	Rule/Heuristic-based											8	8
[OCBZ09]	metric-based	✓	✓									0	2
[BBEAM10]	Machine learning-based					✓						0	1
[MGDM10]	Rule/Heuristic-based	✓					✓		✓			1	4
[MHB10]	metric-based		✓									0	1
[Ram10]	Rule/Heuristic-based	✓								✓		3	5
[OKAG10]	metric-based	✓										0	1
[MGvS10]	metric-based											11	11
[CMRT10]	Rule/Heuristic-based											4	4
[TC11]	Rule/Heuristic-based					✓						0	1
[KVGS11]	Machine learning-based	✓					✓		✓			0	3

Table 2.8: Smell Detection Methods and supported smells

References	Detection Method	God class	Feature envy	Shotgun surgery	Data class	Long method	Functional decomposition	Refused bequest	Spaghetti code	Divergent Change	Long Parameter List	Other smells	Total smells
[BGvS11]	metric-based											7	7
[FBA11]	metric-based	✓			✓		✓					2	5
[TK11]	Rule/Heuristic-based											6	6
[AHTM11]	Rule/Heuristic-based											8	8
[MAB ⁺ 12b]	Machine learning-based	✓					✓		✓			1	4
[FM13]	metric-based					✓		✓			✓	10	13
[SA13]	metric-based	✓										2	3
[ADPAG13]	Rule/Heuristic-based											17	17
[DPXT13]	metric-based	✓										0	1
[LCCY13]	Rule/Heuristic-based							✓				0	1
[VMDP14]	metric-based	✓	✓	✓	✓			✓				5	10
[SKBD14]	Optimization-based	✓	✓		✓		✓		✓		✓	1	7
[PPF ⁺ 14]	metric-based	✓		✓						✓		0	3
[KKS ⁺ 14]	Optimization-based	✓	✓	✓	✓		✓		✓		✓	0	7
[PDMG14]	Rule/Heuristic-based											8	8
[APFC15]	metric-based	✓		✓		✓						0	3
[GEBK15]	Optimization-based	✓			✓		✓					0	3
[CMC15]	Machine learning-based	✓										0	1

Table 2.8: Smell Detection Methods and supported smells

References	Detection Method	God class	Feature envy	Shotgun surgery	Data class	Long method	Functional decomposition	Refused bequest	Spaghetti code	Divergent Change	Long Parameter List	Other smells	Total smells
[PBDP ⁺ 15]	History-based	✓	✓	✓						✓		1	5
[FSMS15]	metric-based					✓						0	1
[FS15]	History-based			✓						✓		1	3
[OKKI15]	Optimization-based											5	5
[Non15]	metric-based		✓									0	1
[ABT15]	Rule/Heuristic-based	✓										0	1
[PPDL ⁺ 16]	Rule/Heuristic-based	✓	✓			✓						2	5
[VVDP ⁺ 16]	metric-based	✓	✓	✓	✓			✓				5	10
[SMT16]	Rule/Heuristic-based	✓	✓		✓	✓		✓		✓	✓	23	30
[MKMD16]	Machine learning-based	✓	✓		✓		✓		✓			0	5

Figure 2.4 shows number of studies detecting a specific smell sorted by the number of studies detecting the smells (the top 20 most frequently detected smells). The figure shows that god class smell has been detected the most in the smells literature. On the other hand, some of the smells have been detected only by one study; these smells include parallel inheritance hierarchy [PBDP⁺15], closure smells [FM13], ISP violation [Mar05], hub-like modularization [SMT16], and cyclic hierarchy [SMT16]. Obviously, there are many other smells that have not been detected by any study. The importance and relevance of a smell cannot be determined by its popularity. Hence, the research community also needs to explore the relatively less commonly detected smells and strengthen the quality analysis.

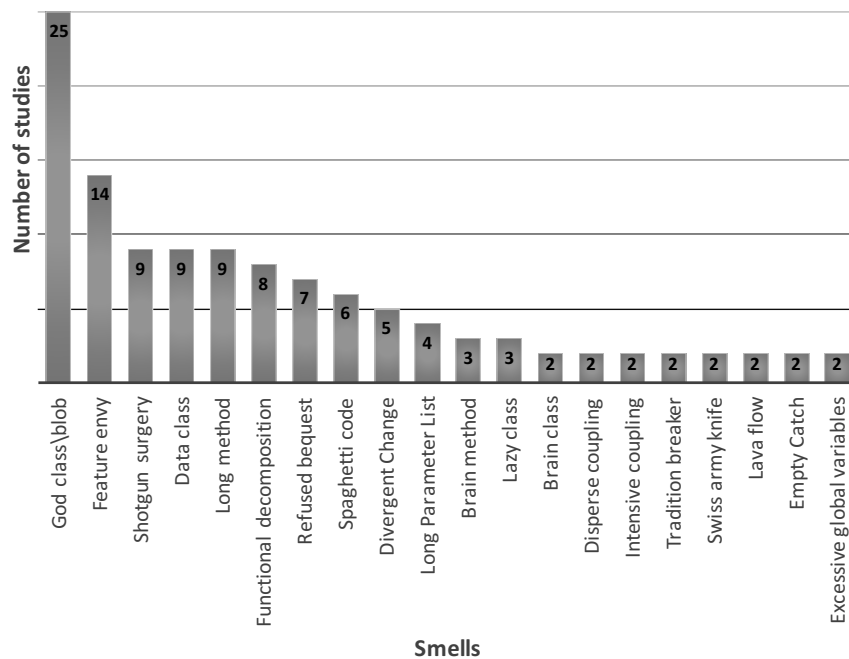
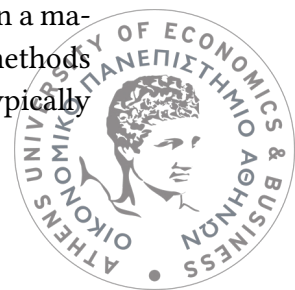


Figure 2.4: The number of studies detecting a specific smell

Further, academic researchers have concentrated heavily on a single programming language, namely Java [RA15]. The 46 smell detection methods for source code shown in Table 2.8 have their targets distributed as follows: 31 for Java, six for models, two for C, two for C++, two for JavaScript, one for C#, and one each for XML and REST APIs. Expanding the smell detection tools to support a wide range of known smells and diverse programming languages and platforms is another open opportunity.

- 3 Immature application of machine learning techniques for smell detection:** Recently many researchers attempted machine learning techniques to detect smells [KVGS09, KVGS11, MAB⁺12b, MAB⁺12a]. However, machine learning techniques to detect smells are considered far from mature. In a recent study, Di Nucci *et al.* [NPT⁺18] note that the problem of detecting smells requires much more research to attain a maturity. There are two shortcomings in the current usage of machine-learning methods for detecting smells. First, heavy feature engineering – current techniques typically



use traditional code metrics as the feature-set. Some authors use customized metrics such as distance metrics used by Liu *et al.* [LXZ18]. Therefore, they need an external tool to compute the set of metrics for the target programming language. If one has a metrics computation tool, she may deduce many smells by combining these metrics [Mar04, SS18] and thus applying a machine-learning method is redundant especially when the human factors and context have not been taken into account during the training process. Second and more importantly, existing attempts limit the machine learning algorithm to use only the metrics that are fed as feature-set. The main premise of using machine-learning method is to bring context and human factor into consideration. However, feeding an algorithm with only metrics, that does not cover either the context or the human factors, defies the purpose of applying machine learning algorithms.

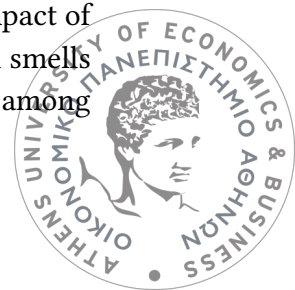
4 Inconsistent smell definitions and detection methods: The abundance of the smell literature has produced inconsistencies in the definition of smells and their detection methods. For example, god class is one of the most commonly researched smells; however, researchers have defined it differently. Riel [Rie96] has defined it as the class that tend to centralize the knowledge in the system. On the other hand, Gabriela *et al.* [CMC15] defined it as a class that has too many methods and Mazeiar *et al.* [SLT06] specified it as the class which is used more extensively than others.

Similarly, based on their description and interpretation, their detection methods also differ significantly and they detect smells inconsistently. Furthermore, in some cases, identical interpretation of smells may also produce different results due to the variation in chosen thresholds of employed metrics.

Even further, metrics tools show inconsistent results even for well-known metrics such as LCOM, CC, and LOC. For example, one tool might implement one variation of LCOM and another tool may realize another or custom variation of the metric while both the tools refer to the metric with the same name. Such inconsistencies in smell definition and their detection methods have been identified by the community [RA15, AFBZ12, SSSG13].

It is, therefore, important and relevant to establish a standard with respect to smell definition, their implementation, as well as commonly used metrics.

5 Impact of smells on productivity: In Section 2.3.3, we present the available literature that discusses the impact of smells on software quality as well as processes and people. It is believed that smells affect mainly maintainability and poor maintainability in turn impacts productivity of the development team. As shown in Section 2.3.3, the current literature draws connection between impact of smells and maintainability. However, the impact of smells on productivity is not yet explored to a sufficient detail. Other researchers [ZHB11] have also identified the need to better understand the impact of smells. We believe that establishing an explicit and concrete relation between smells and productivity will enhance the adoption of the concepts concerning smells among



practitioners.



Implications

In the above discussion, we elaborated on the inherent deficiencies in the present set of smell detection methods. These deficiencies include lack of context and a small number of detectable smells on a very small number of platforms. This analysis clearly calls for effective and widely-applicable smell detection tools and techniques. Inconsistent smell definitions and detection methods indicate the need to set up a standard for smell definitions as well as verified datasets of smells.

2.4 Conclusions

This survey presents a synthesized and consolidated overview of the current knowledge in the domain of software smells. We extensively searched a wide range of conferences and journals for the relevant studies published from year 1999 to 2016. The studies selected in all the phases of the selection, an exhaustive smell catalog, as well as the program that generates the smell catalog are made available online.²

Our study has explored and identified the following dimensions concerning software smells in the literature.

- We reveal five defining characteristics of software smells: *indicator*, *poor solution*, *violates best practices*, *impacts quality*, and *recurrence*.
- We identify and catalog a wide range of smells (more than 250 at the time of writing this thesis) that we made available online and classify them based on 14 focus areas.
- We classify existing smell classifications into four categories: *effect-based*, *principle-based*, *artifact characteristic-based*, and *granularity-based*.
- We curate ten factors that cause smells to occur in a software system. We also classify these causes based on their actors.
- We categorize existing smell detection methods into five groups: *metric-based*, *rules/heuristic-based*, *history-based*, *machine learning-based*, and *optimization-based*.
- We observe that the existing literature does not differentiate between a smell (as an indicator) and a definite quality problem.

²[https://github.com/tushartushar/smells, smells/](https://github.com/tushartushar/smells,smells/)

<http://www.tusharma.in/>





Research opportunities

We identify the following gaps and research opportunities in the present set of tools and techniques.

- The community believes that the existing smell detection methods suffer from high false-positive rates. Also, existing methods cannot define, specify, and capture the context of a smell.
- The currently available tools can detect only a very small number of smells. Further, most of the tools largely only support the Java programming language.
- The machine learning mechanism used to detect smells do not exploit the power of the machine/deep learning are considered far from the maturity.
- Existing literature has produced inconsistent smell definitions. Similarly, smell detection methods and the corresponding produced results are highly inconsistent.
- The current literature does not establish an explicit connection between smells and their impact on productivity of a software development team.



Chapter 3

Methodology

*Research formalizes curiosity;
methodology formalizes research ... in a context.*

In this chapter, we shed light on the thesis objectives, define scope of each subsequent experiment by specifying research questions, and provide an overview of the study design. We first present the study design for production source code, then we elaborate on the study to detect smells using deep learning. Further, we describe our study design for maintainability analysis for configuration code and database schema code.

3.1 Research Objectives

The problem of maintainability analysis for traditional production source code and other sub-domains of software needs to be broken into smaller experiments in order to do justice to each aspect of the larger investigation. In the next sub-sections, we elaborate on the individual experiments that helped us realize the bigger goal.

To separate the discussion, each research question is prefixed with a one or two letter acronym. For research questions investigating maintainability in production source code, we put *P* as prefix. Similarly, we use *D* for deep learning, *C* for maintainability analysis for configuration code, and *DB* for database schema quality analysis related research questions.

3.1.1 Maintainability Analysis for Production Source Code

The first experiment concerns maintainability analysis for production code belonging to a mainstream programming language *i.e.*, C#. It involves analyzing source code and revealing



aspects such as distribution of architecture, design, and implementation smells, relationships such as correlation and collocation among the smells at different granularities, and the relationship between project size and corresponding quality issues.

We formulated the following research questions towards the quality analysis goal of C# projects.

P-RQ1. *What is the distribution of implementation, design, and architecture smells in C# code?*

We investigate the distribution of smells to find out whether there exists a set of implementation, design, and architecture smells that is more prevalent in the analyzed open-source repositories. The answer to this research question may caution the developers about a set of smells expected to have more chances of occurrence and prompts them to take precautionary measures.

P-RQ2. *Do the detected smell instances belonging to different granularities correlate?*

We explore the correlation between smell instances arising at different granularities. Specifically, we explore correlation between design and implementation, as well as architecture and design smell instances. A strong correlation between kinds of code smells would encourage us to understand the occurrence patterns and provide valuable insights into the similarity between these pairs.

Further, we also investigate the correlation between individual design smells and architecture smells. This would help us to find out whether there exist specific types of design smells that are strongly correlated to architecture smells.

P-RQ3. *Is the principle of coexistence applicable to smells in C# projects?*

It is commonly believed that patterns (and smells) co-exist [BMR⁺96a, SSS14] *i.e.*, if we find one smell, it is very likely that we will find many more smells around it. We investigate the intra-category co-occurrences of a smell with other smells to find out whether and to what degree the folklore is true.

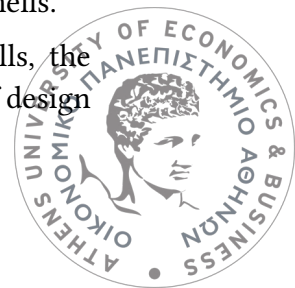
P-RQ4. *Does smell density depend on the size of the C# repository?*

It is commonly believed that the complexity of a software system increases with the size of the system. We investigate the relationship between the size of a C# repository and associated smell density to find out how the smell density changes as the size of a C# project increases. Smell density is a normalized metric that represents the average number of smells identified per thousand lines of code.

P-RQ5. *Are architecture smells collocated with design smells?*

The question aims to explore whether architecture and design smell occur at the same location (*i.e.*, classes) within the source code. A positive result of the collocation analysis would establish a strong relationship between architecture and design smells.

Apart from exploring collocation cumulatively between both kinds of smells, the question investigates the collocation relationships between individual pairs of design



smells and architecture smells. This would help us to figure out whether and to what extent specific design smells show collocation with architecture smells.

P-RQ6. *Can the refactoring of design smells lead to fewer architecture smells?*

In this research question, we figure out the impact of design smell refactorings on architecture smells. It will reveal the degree of influence of design smells on architecture smells. A high influence will hint that by refactoring design smells we can remove a high number of architecture smells. On the other hand, a low influence of architecture smells will lead us to conclusion that we need to put effort to refactor smells at each granularity separately.

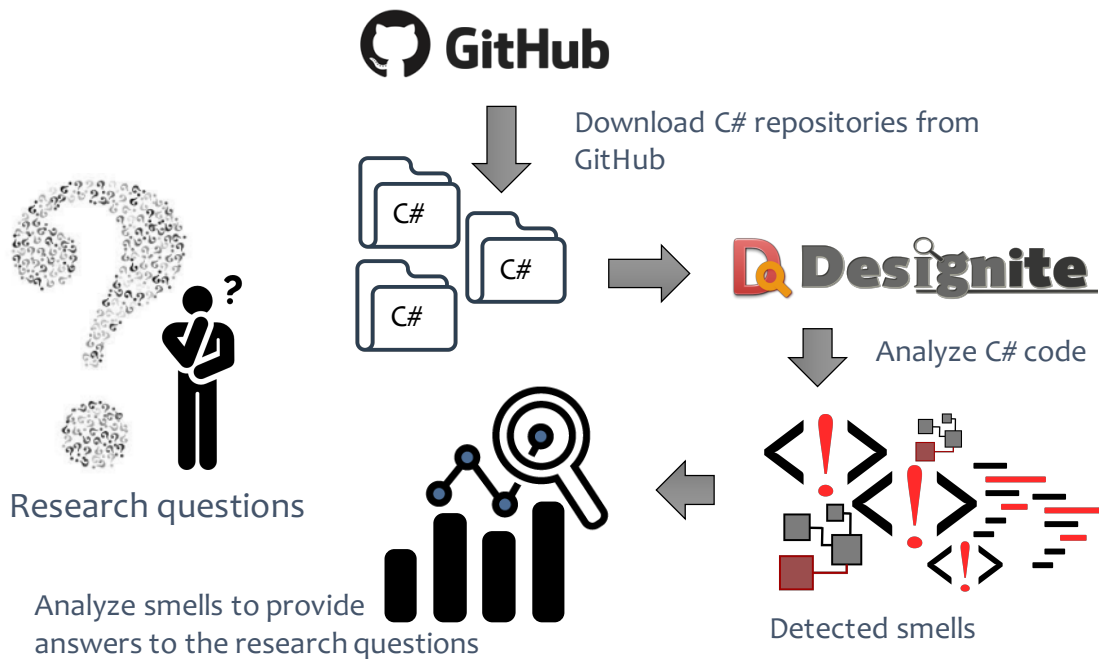
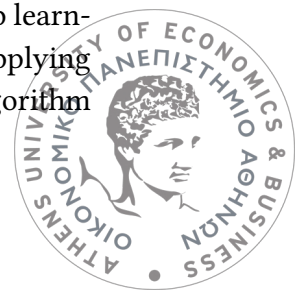


Figure 3.1: Overview of the maintainability analysis study on C# code

Figure 3.1 shows the overview of the experiment. We define a set of aforementioned research questions, select a set of repositories containing C# code by using RepoReapers [MKCN17] and download them. We use Designite [SMT16] to analyze the downloaded repositories and to detect implementation, design, and architecture smells. We analyze the detected smells and perform distribution, correlation, and collocation analysis to address the research questions.

3.1.2 Detecting Smells using Deep Learning

The goal of this research is to explore the possibility of applying state-of-the-art deep learning methods to detect smells. Further, this work inquires into the feasibility of applying transfer-learning. Transfer-learning refers to the technique where a learning algorithm



exploits the commonalities between different learning tasks to enable knowledge transfer across the tasks [BCV13]. Based on the above stated goals, we define the following research questions to explore in this work.

D-RQ1. *Is it possible to use deep learning methods to detect code smells? If yes, which deep learning method performs superior?*

We use CNN and RNN models in this exploration. For the CNN-based architecture, we provide input samples in 1D and 2D format to observe the difference in their capabilities due to the added dimension; we refer to them as CNN-1D and CNN-2D respectively. In the context of this research question, we define the following hypotheses.

D-RQ1.H1 *It is feasible to detect smells using deep learning methods.*

The considered deep learning models are powerful mechanisms that have the ability to detect complex patterns with sufficient training. These models have demonstrated high performance in the domain of image processing [KSH12, SLJ⁺15] and natural language processing [LPM15]. We believe we can leverage these models in the presented context.

D-RQ1.H2 *CNN-2D performs better than CNN-1D in the context of detecting smells.*

The rationale behind this hypothesis is the added dimensionality in CNN-2D. The 2D model might observe inherent patterns when input data is presented in two dimensions that may possibly be hidden in one dimensional format. For instance, a 2-D variant could possibly identify the nesting depth of a method easier than its 1-D counterpart when detecting *complex method* smell.

D-RQ1.H3 *An RNN model performs better than CNN models in the smell detection context.*

RNN are considered better for capturing sequential patterns and have the reputation to work well with text. Thus, taking into account the similarities that source code and natural language share, RNN could prove superior than CNN models.

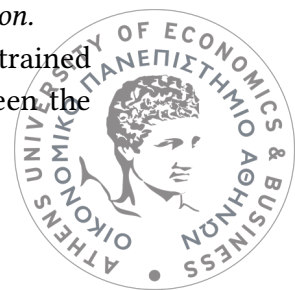
D-RQ2. *Is transfer-learning feasible in the context of detecting smells? If yes, which deep learning model exhibits superior performance in detecting smells when applied in transfer-learning setting?*

Transfer-learning is the capability of an algorithm to exploit the similarities between different learning tasks and offering a solution of a task by transferring knowledge acquired while solving another task. We would like to explore whether it is feasible to train a deep learning model from samples of C# and predict the smells using this trained model in samples of Java programming language.

We derive the following hypotheses.

D-RQ2.H1 *It is feasible to apply transfer-learning in the context of code smell detection.*

We train the deep learning models using C# code fragments and evaluate the trained model using Java fragments. Given the high similarity in the syntax between the



two programming languages, we believe that we may train the model from training samples and use the trained model to classify smelly and non-smelly fragments from our evaluation samples.

D-RQ2.H2 *Transfer-learning performs inferior compared to direct-learning.*

Direct-learning in the context of our study refers to the case where training and evaluation samples belong to the same programming language. We expect that the performance of the models in the transfer-learning could be inferior to that compared to direct-learning given both the problems are equally hard *i.e.*, negative and positive sample showing similar distribution.

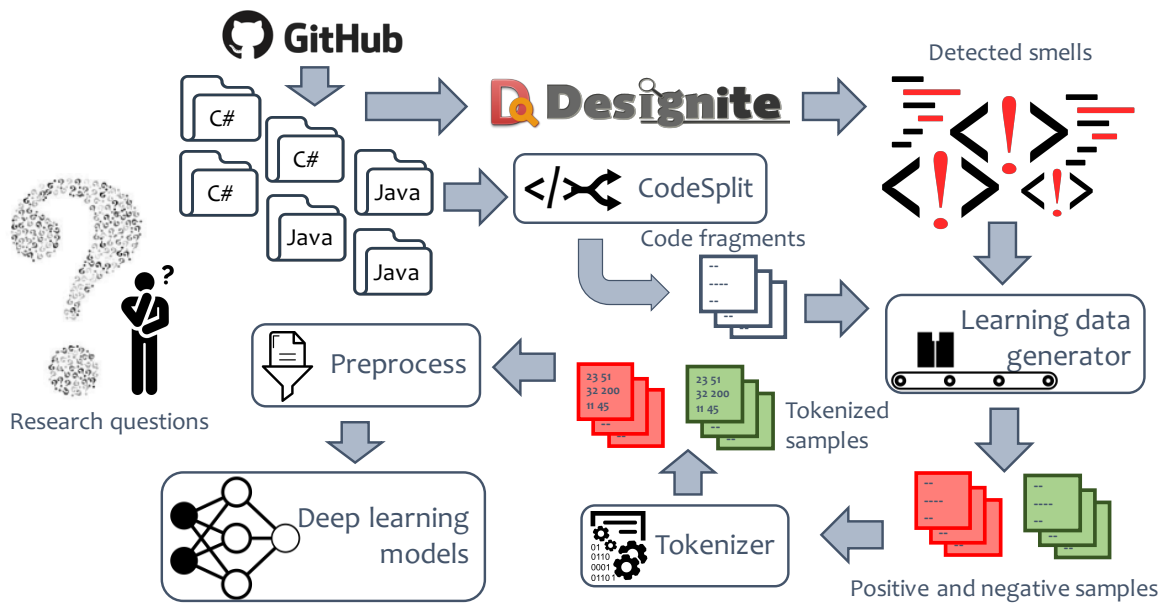


Figure 3.2: Overview of the Proposed Method

Figure 3.2 provides an overview of the experiment. We download 1,072 C# and 100 Java repositories from GitHub. We use Designite and DesigniteJava to analyze C# and Java code respectively. We use CodeSplit to extract each method and class definition into separate files from C# and Java programs. Then the learning data generator uses the detected smells to bifurcate code fragments into positive or negative samples for a smell—positive samples contain the smell while the negative samples are free from that smell. Tokenizer takes a method or class definition and generates integer tokens for each token in the source code. We apply preprocessing operation, specifically duplicates removal, on the output of Tokenizer. The processed output of Tokenizer is ready to feed to the neural networks.

3.1.3 Maintainability Analysis for Configuration Code

In the pursuit to extend the maintainability analysis, we carry out a study to analyze the existing configuration code and evaluate the associated code quality to examine the existing practices towards keeping configuration code maintainable.



We formulated the following research questions towards the quality analysis goal of configuration code.

C-RQ1. *What is the distribution of maintainability smells in configuration code?*

We investigate the distribution of configuration smells to find out whether there exists a set of implementation and design configuration smells that occur more frequently with respect to another set of configuration smells.

C-RQ2. *What is the relationship between the occurrence of design configuration smells and implementation configuration smells?*

We study the instances of design configuration smells and implementation configuration smells to discover the degree of co-occurrence between the two categories of configuration smells.

C-RQ3. *Is the principle of coexistence applicable to smells in configuration projects?*

In traditional software engineering, it is said that patterns (and smells) co-exist as “No pattern is an island” [BMR⁺96b] i.e. if we find one, it is very likely that we will find many more around it [BMR⁺96b, SSS14]. We investigate the intra-category co-occurrence of a smell with other smells to find out whether the folklore is true in the context of configuration smells. Furthermore, we investigate whether all the smells in each of the categories follow the principle with a same degree.

C-RQ4. *Does smell density depend on the size of the configuration project?*

We investigate the relationship between the size of a configuration project and associated smell density for both smell categories to find out how the smell density changes as the size of the configuration project increases.

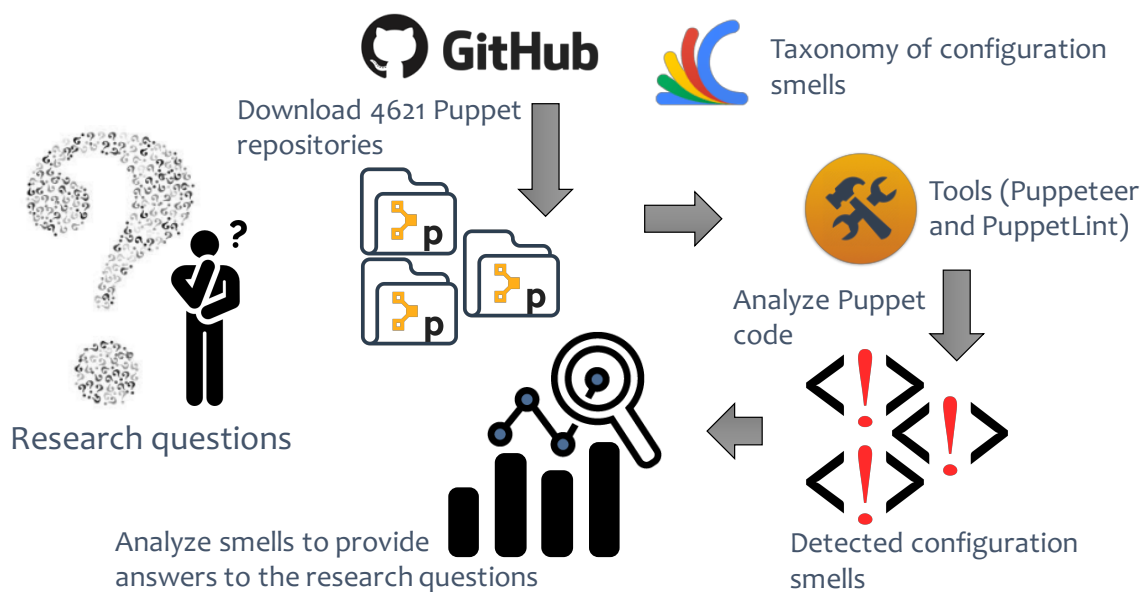


Figure 3.3: Overview of the maintainability analysis study on configuration (Puppet) code



Figure 3.3 provides an overview of the maintainability analysis study on configuration smells. We collate best practices followed in IaC domain and define a taxonomy of configuration smells. To detect the majority of implementation configuration smells, we use Puppet-Lint [Pup16c]. Due to the lack of an existing tool that can detect design configuration smells, we develop a tool namely *Puppeteer* for detecting the cataloged design configuration smells. We identify repositories containing Puppet code and download them from GitHub [Git16]. We download 4,621 repositories containing 142,662 Puppet files and 8.9 million lines of code and analyze them with the help of Puppet-Lint and Puppeteer. We grouped the information collected based on the data required to answer the research questions and deduce our observations.

3.1.4 Maintainability Analysis for Database Code

Further, we set out a study to understand database code quality by mining database schema smells and explore their relationship with other software artifacts. The chosen subject systems are a wide variety of industrial as well as open-source software systems. We keep the focus of the study on performance and maintainability quality attributes of relational database code. Characteristics of smells, such as frequency (or the occurrence pattern) of smells [HPvD12, LVKM⁺14, BQO⁺12], provide dimensions of prioritization and refactoring. Similarly, relationships of smells with domains, frameworks, and other application characteristics [LVKM⁺14, FFM⁺13], help us understand the interplay of smells with application characteristics. In the context of database programming, ORM (Object-Relational Mapping) frameworks simplify database access by providing an abstraction. However, it is not understood whether the usage of an ORM framework in an application will lead us to fewer number of smells. Further, studying co-occurrence of database schema smells will complement the existing studies exploring properties of co-occurrence among smells [MVL03, SFS16].

With this background, we explore the following research questions.

DB-RQ1. *What are the occurrence patterns of database smells?*

We examine the distribution of database smells to find out whether there exists a set of database smells that occurs more frequently in general than another set of database smells.

DB-RQ2. *Does the size of the project or the database play a role in smell density?*

We investigate the relationship of the size of the project (both the total lines of code as well as total number of CREATE TABLE statements) and smell density.

DB-RQ3. *Does the nature of code (type of the application, or usage of ORM frameworks) affect the smell density?*

The usage of an ORM framework makes it easier to work with a database. We explore whether the usage of ORM frameworks and the type of the application influence database smell density.

DB-RQ4. *What is the degree of co-occurrence among database smells?*

Patterns and smells tend to occur together [BMR⁺96a, SSS14]. We examine the degree



of co-occurrence among database smells to find out a set of database smells that is likely to occur when a database smell gets detected.

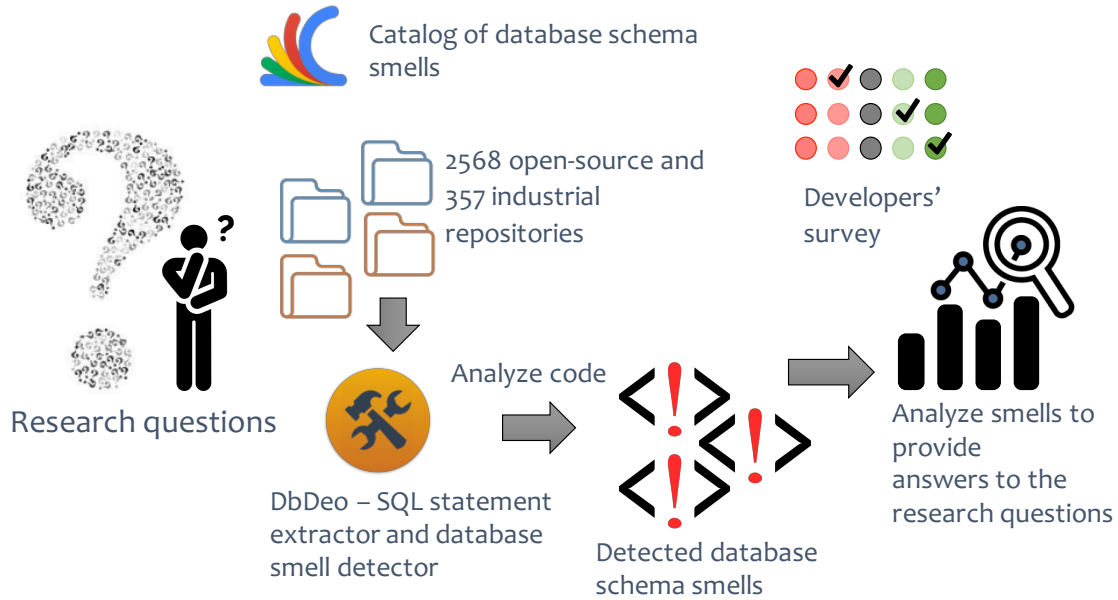


Figure 3.4: Overview of the maintainability analysis study on database schema code

Figure 3.4 provides an overview of the experiment. To study the addressed questions, we compiled a catalog of 13 database schema smells. We attempt to understand developers' perspective on database schema smells through an online survey. We developed a tool viz. *DbDeo* to extract embedded SQL statements from host source code (in which the SQL statements are embedded) and identify cataloged database smells. We analyzed 2,568 GitHub open-source repositories and 357 industrial close-source repositories containing SQL statements and provide empirical answers to each of the posed research questions.

3.2 Theoretical Background

In this section, we present the necessary theoretical background on which the experiments in this thesis are based on. We define and specify each category of smells – code smells (at three granularities – implementation, design, and architecture), configuration smells, and database schema smells and elaborate on the challenges of applying a deep learning-based smell detection mechanism.

3.2.1 Code Smells

The term *code smells* is an umbrella term; depending upon the granularity, scope, and impact, code smells can be classified as implementation, design, and architecture smells [SSS16].



3.2.1.1 Architecture Smells

Smells arising at architecture granularity (typically perceived at component level) and affecting software quality at system-level are referred as architecture smells. In this thesis, we examine, detect, and analyze seven common architecture smells. We provide their definitions below.

1. **Cyclic Dependency:** This smell arises when two or more architecture components depend on each other directly or indirectly [MCKX15, LR06].
2. **Unstable Dependency:** This smell arises when a component depends on other less stable components [FDW⁺16]. Stable Dependencies Principle (SDP) [Mar02] states that the dependencies between packages should be in the direction of the stability of the packages. Hence, a package should only depend on packages that are more stable than it is.
3. **Ambiguous Interface:** This smell arises when a component offers only a single, general entry-point into the component [GPEM09]. This smell typically appears in event-based publish-subscribe systems where interactions are not explicitly modelled and multiple components exchange event messages via a shared event bus.
4. **God Component:** This smell occurs when a component is excessively large either in terms of LOC (Lines Of Code) or number of classes [LR06].
5. **Feature Concentration:** This smell occurs when a component realizes more than one architectural concerns or features [dAAC14b]. In other words, the component is not cohesive.
6. **Scattered Functionality:** This smell arises when multiple components are responsible for realizing the same high-level concern [GPEM09]. It is an indication that possibly classes or methods must be moved from one component to another in order to reduce coupling among components and enhance cohesion within each component.
7. **Dense Structure:** This smell arises when components have excessive and dense dependencies without any particular structure [SFS16].

3.2.1.2 Design Smells

Design smells are structures in the design that indicate violation of fundamental design principles and negatively impact design quality [SSS14]. Table 3.1 lists all the design smells [SSS14] considered in this work along with their brief descriptions.

3.2.1.3 Implementation Smells

The scope and granularity of implementation smells is limited to typically a method. Table 3.2 lists all the implementation smells taken into consideration in this study.



Table 3.1: Description of Detected Design Smells

Design smell	Brief description
Broken Hierarchy	a supertype and its subtype conceptually do not share an “IS-A” relationship
Broken Modularization	data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions
Cyclically-dependent Modularization	two or more abstractions depend on each other directly or indirectly
Cyclic Hierarchy	a supertype in a hierarchy depends on any of its subtypes
Deep Hierarchy	an inheritance hierarchy is “excessively” deep
Deficient Encapsulation	the declared accessibility of one or more members of an abstraction is more permissive than actually required
Duplicate Abstraction	two or more abstractions have identical names or identical implementation
Hub-like Modularization	an abstraction has high incoming and outgoing dependencies
Imperative Abstraction	an operation is turned into a class
Insufficient Modularization	an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, or implementation complexity
Missing Hierarchy	a code segment uses conditional logic to explicitly manage variation in behaviour
Multifaceted Abstraction	an abstraction has more than one responsibility assigned to it
Multipath Hierarchy	a subtype inherits both directly as well as indirectly from a supertype
Rebellious Hierarchy	a subtype rejects the methods provided by its supertype(s)
Unexploited Encapsulation	client code uses explicit type checks
Unfactored Hierarchy	there is unnecessary duplication among types in a hierarchy
Unnecessary Abstraction	an abstraction that is actually not needed
Unutilized Abstraction	an abstraction is left unused
Wide Hierarchy	an inheritance hierarchy is “too” wide

3.2.2 Exploring Deep Learning-based Solution for Smell Detection

In this section, we present challenges in applying deep learning techniques on source code as well as selection of code smells for our exploration.



Table 3.2: Description of Detected Implementation Smells and Their Distribution

Implementation smell	Brief description
Complex Conditional	a complex conditional statement
Complex Method	a method with high cyclomatic complexity
Duplicate Code	a code clone within a method
Empty Catch Block	a catch block of an exception is empty
Long Identifier	an identifier with excessive length
Long Method	a method is excessively long
Long Parameter List	a method has long parameter list
Long Statement	an excessive long statement
Magic Number	an unexplained number is used in an expression
Missing Default	a switch statement does not contain a default case
Virtual Method Call from Constructor	a constructor calls a virtual method

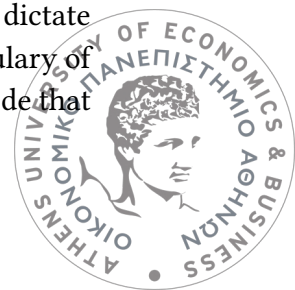
3.2.2.1 Challenges in Applying Deep Learning on Source Code

Applying deep learning techniques on source code is non-trivial. In this section, we present challenges that we face in the process of applying deep learning techniques on source code.

Analogy with other problems:

Deep learning is advancing rapidly in domains that address problems of image, video, audio, text, and speech processing [LBH15]. Consequently, these advances drive current trends in deep learning and inspire applications across disciplines. As such, studies that apply deep learning on source code rely heavily on results from these domains, and particularly that of text mining.

Based on prior observations that demonstrate similarity between source code and natural language [HBS⁺12], the research community has largely addressed relevant problems on mining source code by adopting latest state-of-the-art natural language processing methods [APS16, PDDL⁺16, IKCZ16, VCD17, YN17]. However, besides similarities, there also exist major differences that need to be taken into consideration when designing such studies. First of all, source code, unlike natural language, is semantically brittle; minor syntactic changes can drastically change the meaning of code [ABDS18]. As an effect, treating code as text by ignoring the underlying formal semantics carries the risk of not preserving the appropriate meaning. Besides formal semantics, the syntax of source code obviously presents substantial differences compared to the syntax found in text. As a result, methods that perform well on text are likely to under-perform on source code. Architectures involving CNN-1D layers, for instance, have been proven effective for matching subsequences of short lengths [Cho17], which are often found in natural language where the length of sentences is limited. This however does not necessarily apply on self-contained fragments of source code, such as method definitions, which tend to be longer. Finally, even though good practices dictate naming conventions in coding, unlike natural language, there is no universal vocabulary of source code. This results to a diversity in the artificial vocabulary found in source code that



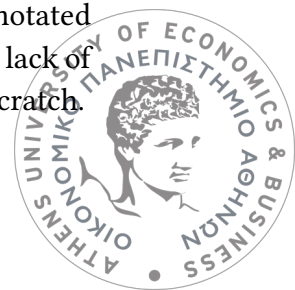
may affect the quality of the models learned.

Approaches that treat code as text mainly focus on the mining of sequential patterns of source code tokens. Other emerging approaches look into structural characteristics of the code with the objective of extracting visual patterns delineated on code [OAH⁺18]. Even though there are features in source code, such as nesting, which demonstrate distinctive visual patterns, treating source code in terms of such patterns and ignoring the rich intertwined semantics carries the risk of oversimplifying the problem.

Lack of resources:

Research employing deep learning techniques on software engineering data, including source code as well as other relevant artifacts, is still young. Consequently, results against traditional baseline techniques are very limited [FM17, HD17]. Especially when it comes to processing solely source code artifacts, relevant studies are scarce and mostly address the problem of drawing out semantics related to the functionality of a piece of code [APS16, WVLVP15, WTV16, MLZ⁺16, PHN⁺15]. To the best of our knowledge, our study is the first to thoroughly investigate the application of deep learning techniques with the objective of examining characteristics of source code quality. Therefore, a major challenge in studies of this kind is that there is no prior knowledge that would guide this investigation, a challenge reflected on all stages of the inquiry. At the level of designing an experiment, there exist no rules of thumb indicating a set up for a deep learning architecture that adequately models the fine-grained features required for the problem in hand. Furthermore, at the level of training a model, there is no prior baseline for hyper-parameters that would lead to an optimal solution. Finally, at the level of evaluating a trained model, there exist no benchmarks to compare against; there is no prior concrete indication on the expected outcomes in terms of reported metrics. Hence, a result that would appear sub-optimal in another domain such as natural language processing, may actually account for a significant advance in software quality assessment.

Besides challenges that relate to the know-how of applying deep learning techniques on source code, there are technical difficulties that arise due to the paucity of curated data in the field. The need for openly available data that can serve for replicating data-driven studies in software engineering has been long stressed [Rob10]. The release of curated data in the field is encouraged through badging artifact-evaluated papers as well as dedicated data showcase venues for publication. However, the software engineering domain is still far from providing benchmark datasets, whereas the available datasets are limited to curated collections of repositories with associated metadata that lack ground truth annotation that is essential for a multitude of supervised machine learning tasks. Therefore, unlike domains such as image processing and natural language processing where an abundance of annotated data exist [KH09, DDS⁺09, LCB10, MDP⁺11], in the field of software engineering the lack of gold standards induces the inherent difficulty of collecting and curating data from scratch.



3.2.2.2 Selection of Smells

Over the last two decades, the software engineering community has documented many smells associated with different granularities, scope, and domains [SS18]. A comprehensive taxonomy of the software smells can be found online.¹ For this study, selection of smells is a crucial decision. The scope of the higher granularity smells, such as design and architecture smells, is large, often spanning to multiple classes and components. It is essential to provide all the intertwined source code fragments to the deep learning model to make sure that the model captures the key deciding elements from the provided input source code. Hence, it is naturally difficult to detect them using deep learning approaches, unless extensive feature engineering is performed beforehand in order to attain an appropriate representation of the data. We started with implementation smells because they can be detected typically just by looking at a method. However, we would like to avoid very simple smells (such as *long method*) which can be easily detected by less sophisticated techniques.

We chose *complex method* (CM—i.e., the method has high cyclomatic complexity), *magic number* (MN—i.e., an unexplained numeric literal is used in an expression), and *empty catch block* (ECB—i.e., a catch block of an exception is empty). These three smells represent three different kinds of smells where neural networks have to spot specific features. For instance, to detect *magic number*, the neural networks must spot a specific range of tokens representing magic numbers. On the other hand, detection of *complex method* requires looking at the entire method and the structural property within it (i.e., nesting depth of the method). For the detection of *empty catch block* the neural network has to recognize a sequence of a try block followed by an empty catch block.

To expand the horizon of the experiment, we also select *multifaceted abstraction* (MA—i.e., a class has more than one responsibility assigned to it) design smell. The scope of this smell is larger (i.e., the whole class) and detection is not trivial since the neural network has to capture cohesion aspect (typically captured by the Lack of Cohesion of Methods (LCOM) metric in deterministic tools) among the methods to detect it accurately. This smell not only allows us to compare the capabilities of neural networks in detecting implementation smells with design smells but also sets the stage for the future work to build on.

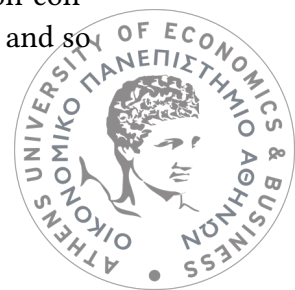
3.2.3 Configuration Smells

We define configuration smells as follows:

Configuration smells are the characteristics of a configuration program or script that violate the recommended best practices and potentially affect the program's quality in a negative way.

Similar to traditional software engineering practices where smells are classified based on granularity and scope, configuration smells are also classified as implementation configuration smells, design configuration smells, documentation configuration smells, and so

¹<http://www.tusharma.in/smells>



on. In this work, our focus is on two major categories of configuration smells namely implementation configuration smells and design configuration smells.

3.2.3.1 Implementation Configuration Smells

Implementation configuration smells are quality issues such as naming convention, style, formatting, and indentation in configuration code. We prepare a list of recommended best practice by studying available resources, such as the Puppet style guide [Sty16] and rules implemented by Puppet-Lint. We group the best practices based on their similarity and arrive at a corresponding implementation configuration smell when a best practice is violated. Table 3.3 lists the implementation configuration smells and corresponding set of best practices.

Here, we present a list of implementation configuration smells with a brief description. Figure 3.5 shows an annotated Puppet example with all the cataloged implementation configuration smells.

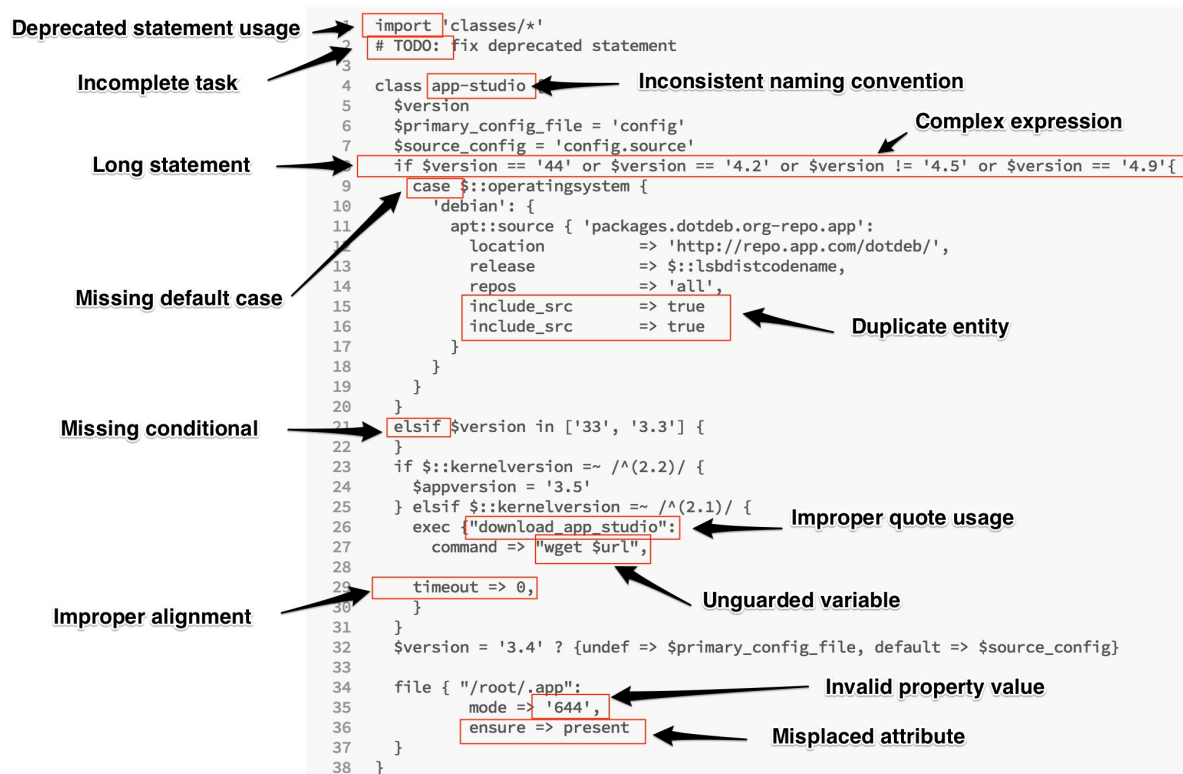


Figure 3.5: An annotated Puppet example with all the cataloged implementation configuration smells

1. **Missing Default Case (IMD)** A default case is missing in a *case* or *selector* statement.
2. **Inconsistent Naming Convention (INC)** The used naming convention deviates from the recommended naming convention.



Table 3.3: Mapping Between Implementation Configuration Smells and Corresponding Best Practices

Smells	Best practices
Missing default case	Case and Selector statements should have a default case
Inconsistent naming convention	The names of variables, classes and defines should not contain a dash
Complex expression	Expressions should not be too complex
Duplicate entity	Duplicated hash keys and parameters should be removed
Misplaced attribute	<ul style="list-style-type: none"> • “ensure” attribute should be the first attribute specified • The required parameters for a class or ‘define’ should be listed before optional parameters • Right-to-left chaining arrows should not be used
Improper alignment	<ul style="list-style-type: none"> • Properly align arrows (arrows are not all placed one space ahead of the longest attribute) • Tabulation characters should not be used
Invalid property value	<ul style="list-style-type: none"> • “ensure” property of file resource should be valid • File mode should be represented by a valid 4-digit octal value (rather than 3) or symbolically • The path of “puppet:///” URL should start with “modules/”
Incomplete tasks	“FIXME” and “TODO” tags should be handled
Deprecated statement usage	Deprecated node inheritance and “import” statement should not be used
Improper quote usage	<ul style="list-style-type: none"> • Booleans should not be quoted • Variables should not be used in single quoted strings • Unquoted node names should not be used • Resource titles should be quoted • Literal boolean values should not be used in comparison expressions
Long statement	Lines should not be too long
Incomplete conditional	“if ... elsif” constructs shall be terminated with an “else” clause
Unguarded variable	Variables should be enclosed in braces when being interpolated in a string

3. **Complex Expression** (ICE) A program contains a difficult to understand complex expression.

4. **Duplicate Entity** (IDE) Duplicate hash keys or duplicate parameters present in the



configuration code.

5. **Misplaced Attribute** (IMA) Attribute placement within a resource or a class has not followed a recommended order (for example, mandatory attributes should be specified before the optional attributes).
6. **Improper Alignment** (IIA) The code is not properly aligned (such as all the arrows in a resource declaration) or tabulation characters are used.
7. **Invalid Property Value** (IPV) An invalid value of a property or attribute is used (such as a file mode specified using 3-digit octal value rather than 4-digit).
8. **Incomplete Tasks** (IIT) The code has “FIXME” and “TODO” tags indicating incomplete tasks.
9. **Deprecated Statement Usage** (IDS) The configuration code uses one of the deprecated statements (such as “import”).
10. **Improper Quote Usage** (IQU) Single and double quotes are not used properly. For example, boolean values should not be quoted and variable names should not be used in single quoted strings.
11. **Long Statement** (ILS) The code contains long statements (that typically do not fit in a screen).
12. **Incomplete Conditional** (IIC) An “if..elsif” construct used without a terminating “else” clause.
13. **Unguarded Variable** (IUV) A variable is not enclosed in braces when being interpolated in a string.

3.2.3.2 Design Configuration Smells

Design configuration smells reveal quality issues in the module design or structure of a configuration project. Various available sources, such as the Puppet style guide [Sty16], blog entries [Lar16a, Lar16b], and videos of technical talks [Lar16c] highlight the best practices to be followed for configuration code. We obtain a list of commonly occurring design configuration smells from the violation of these best practices at design-level. We assign relevant names (often inspired by the traditional names of smells) to the smells and document their forms representing variations of the smells. Here, we present design configuration smells with a brief description.

1. **Multifaceted Abstraction** (DMF) Each abstraction (e.g. a resource, class, ‘define’, or module) should be designed to specify the properties of a single piece of software. In other words, each abstraction should follow single responsibility principle [Mar02]. An abstraction suffers from *multifaceted abstraction* when the elements of the abstraction are not cohesive.



The smell may occur in the following two forms:

- a resource (file, package, or service) declaration specifies attributes of more than one physical resources, or
 - all the language elements declared in a class, ‘define’, or a module are not cohesive.
2. **Unnecessary Abstraction** (DUA) A class, ‘define’, or module must contain declarations or statements specifying the properties of a desired system. An empty class, ‘define’, or module shows the presence of *unnecessary abstraction* smell and thus must be removed.
 3. **Imperative Abstraction** (DIA) Puppet is declarative in nature. The presence of imperative statements (such as “exec”) defies the purpose of the language. An abstraction containing numerous imperative statements suffers from *imperative abstraction* smell.
 4. **Missing Abstraction** (DMA) Resource declarations and statements are easy to use and reuse when they are encapsulated in an abstraction such as a class or ‘define’. A module suffers from the *missing abstraction* smell when resources and language elements are declared and used without encapsulating them in an abstraction.
 5. **Insufficient Modularization** (DIM) An abstraction suffers from this smell when it is large or complex and thus can be modularized further. This smell arises in following forms:
 - if a file contains a declaration of more than one class or ‘define’, or
 - if the size of a class declaration is large crossing a certain threshold, or
 - the complexity of a class or ‘define’ is high.
 6. **Duplicate Block** (DDB) A duplicate block containing a set of statements more than a threshold indicates that probably a suitable abstraction definition is missing. Thus a module containing such a duplicate block suffers from *duplicate block* smell.
 7. **Broken Hierarchy** (DBH) The use of inheritance must be limited to the same module. The smell occurs when, the inheritance is used across namespaces where inheritance is not natural (“is-a” relationship is not followed).
 8. **Unstructured Module** (DUM) Each module in a configuration repository must have a well-defined and consistent module structure. A recommended structure for a module is the following.
 - *Module name*
 - manifests
 - files



- templates
- lib
- facts.d
- examples
- spec

An ad-hoc structure of a repository suffers from *unstructured module* smell that impacts understandability and predictability of the repository.

9. **Dense Structure** (DDS) This smell arises when a configuration code repository has excessive and dense dependencies without any particular structure.
10. **Deficient Encapsulation** (DDE) This smell arises when a node definition or ENC (External Node Classifier) declares a set of global variables to be picked up by the included classes in the definition.
11. **Weakened Modularity** (DWM) Each module must strive for high cohesion and low coupling. This smell arises when a module exhibits high coupling and low cohesion.

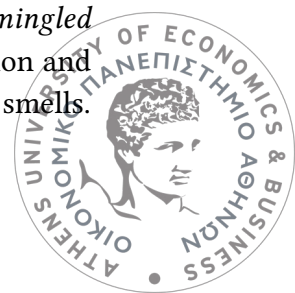
3.2.4 Database Smells

We define database smells as follows:

Database smells are the characteristics of database code (either DDL or DML SQL statements), database system, or stored data that indicate violation of the recommended best practices and potentially affect the quality of the software system in a negative way.

We categorize database smells in three categories to understand them better.

- **Schema smells:** Smells that arise due to poor schema design are classified as database schema smells. Smells presented in this section such as *compound attribute*, *index abuse*, and *god table* are examples of database schema smells.
- **Query smells:** Smells arising from poorly written SQL queries are specified as database query smells. *Misused null* [Kar10] (when null is used as an ordinary value in SQL queries) and *non-grouped column reference* [Kar10] (when a query references at least one non-grouped column in the presence of *group by* clause) are examples of database query smells.
- **Data smells:** Data smells arise from poor data handling in databases. *Intermingled data types* (where numbers and alphabets are intermingled leading to confusion and subtle bugs; for instance, using ‘O’ instead of ‘0’ in 7O34) is an example of data smells.



In this thesis, we focus only on database schema smells. We carry out a comprehensive exploration of resources that discuss best practices as well as common database smells or anti-patterns. We study wide variety of resources including books [Kar10], research literature [Che15, NC15, EV15], industrial white-paper [Red17], and discussions on question-answer sites [dbS10]. We summarize the result of our exploration in the form of a catalog of database schema smells.

1. **CA: Compound attribute:** This smell arises when a column is used to store a non-atomic attribute. For instance, storing comma-separated lists for an attribute to avoid creating an intersection table for a many-to-many relationship [Kar10, Red17] or storing a JSON file which is not used atomically [dbS10].

Each attribute value must be stored and retrieved atomically. If a table does not adhere to this practice, the resultant schema introduces multiple problems. For instance, a user has to write more complex queries (using pattern-matching expressions) to retrieve data from this table. Such complex queries are prone to inaccurate results. Also, such queries cannot exploit available indexes. Even further, these queries are not portable due to vendor specific support to pattern-matching expressions.

2. **AL: Adjacency list:** The smell occurs when an attribute in a table refers another row in the same table *i.e.*, a table has a recursive relationship to model hierarchical structure [Kar10, Red17].

Querying a tree with adjacency list is quite difficult and error-prone. Specifically, deleting a node from a tree which is modelled using adjacency list is non-trivial and prone to introduce errors in the database.

3. **SK: Superfluous key:** This smell arises when an unnecessary superfluous pseudo key is defined in a table where other attribute(s) in the table may serve as a primary key [Kar10].

Choosing an appropriate primary key is an essential requirement for a table. A pseudo key could be defined when the present set of attributes could not serve as a primary key. However, a pseudo key is unnecessary and even erroneous (leads to duplicate rows) when the existing set of attributes of the table could be used as a primary key.

4. **MC: Missing constraints:** This smell arises when constraints for a foreign key are missing from a schema definition [Kar10, Red17].

Referential integrity is an essential property of relational databases. Values referenced in a foreign key column must exist in the columns of primary or unique keys of the parent table. It can be easily achieved by defining constraints on foreign keys. However, when such constraints are missing for a foreign key it leads to compromised referential integrity of the database.

5. **MD: Metadata as data:** This smell occurs when metadata is stored as data in the form of EAV (Entity-Attribute-Value) pattern [Kar10, Red17].



In a relational table, all the attributes are equally applicable for all the rows in the table. It is tempting to implement EAV pattern when a subset of attributes applicable for a subset of rows and the rest of attributes for rest of the rows. However, this arrangement introduces many deficiencies in the database; for example, one can't use native SQL data types (leading to invalid data), enforce referential integrity, or make up attribute names.

6. **PA: Polymorphic association:** This smell occurs when a table uses a multi-purpose foreign key [Kar10, Red17].

Relational database schema does not allow us to declare polymorphic association. However, many times developers define an additional column in a table as a tag to realize a polymorphic association. This arrangement makes it difficult to query the table and compromises readability and understandability.

7. **MA: Multicolumn attribute:** This smell arises when multiple serial columns are created for an attribute [Kar10, dbS10].

In cases when an attribute may have one or more values, it is tempting to create multiple columns for the attribute in a table. However, such a schema design makes querying the table very difficult and verbose.

8. **CT: Clone tables:** This smell occurs when a table is split horizontally in multiple tables using some criterion (for example, year) to achieve scalability [Kar10].

This smell not only makes the querying difficult but also introduces problems managing data integrity.

9. **VA: Values in attribute definition:** This smell arises when specific values are defined in an attribute definition to restrict possible values of the attribute [Kar10].

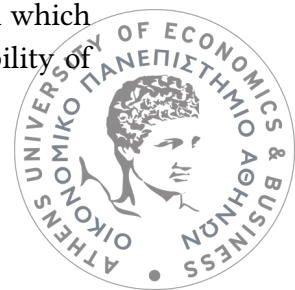
Specifying all possible values for an attribute in schema definition mixes metadata with data which is not recommended. This smell makes it difficult to extend or modify the list of accepted values for an attribute.

10. **IA: Index abuse:** This smell arises when the indexes are used poorly [Kar10, Red17]. This smell has the following variants: 1) Missing indexes 2) Insufficient indexes (indexes must be prepared at least for primary and foreign keys), and 3) Unused indexes

Creating effective indexes is not trivial; it requires judicious planning. A database with a deficient plan for indexes performs poorly.

11. **GT: God table:** This smell arises when a table contains excessive number of attributes [dbS10, Red17].

Excessive number of attributes tend to violate the principles of normalization which in turn introduce a variety of problems. Additionally, it impacts maintainability of the database.



12. **MN: Meaningless name:** This smell occurs when a table or an attribute name is cryptic or meaningless [dbS10].

Meaningless or cryptic names hamper readability of the database's schema.

13. **OA: Overloaded attribute names:** This smell occurs when two or more attributes are defined with identical names but as distinct data types in different tables [Red17].

Identical names with different data types create confusion and could lead to subtle bugs in queries.



Chapter 4

Implementation

Even the best designs are useless without an effective implementation.

In this chapter, we elaborate on the implementation details for each experiment that we carried out. We illustrate the tools, such as smell detection tools, employed for the experiments, detection method for each supported smell, and the realization of qualitative mechanisms.

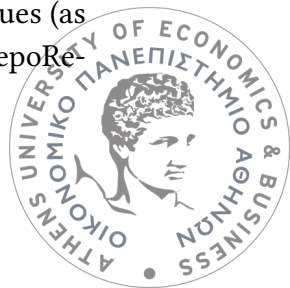
4.1 Analyzing Production Code for Quantitative Maintainability Assessment

In this section, we illustrate the process we adopted to select, download, and mine C# repositories. We also discuss the implementation details of Designite — a software design quality assessment tool that we developed to analyze C# code.

4.1.1 Mining C# Repositories

We used the following protocol to identify our subject systems.

- We use RepoReapers [MKCN17] to select a set of repositories as subject systems from GitHub. RepoReapers analyzes GitHub repositories and provides their quality characteristics based on eight dimensions. These dimensions are architecture (as evidence of code organization), continuous integration and unit testing (as evidence of quality), community and documentation (as evidence of collaboration), history, issues (as evidence of sustained evolution), and license (as evidence of accountability). RepoReapers assigns a score corresponding to each dimension.



- We select all the repositories containing C# code where at least six out of eight Re-poReapers' dimensions had suitable scores. We consider a score to be suitable if it has score greater than zero.
- Next, the repositories selected through the above-mentioned criteria are sorted based on the number of assigned stars. We select repositories tagged with more than 10 stars.
- Following these criteria, we download more than 3,400 repositories using our code smell detection and analyze them using our quality analysis tool – Designite. Some of the repositories could not be analyzed due to either missing external dependencies or custom build mechanisms (*i.e.*, missing standard C# project files). We successfully analyze 3,209 repositories for the study.
- Test code contains different types of smells (*viz.* test smells [Deu01]) which is not in the scope of this experiment. Hence, we exclude the test code belonging to the selected software repositories from our empirical analysis.

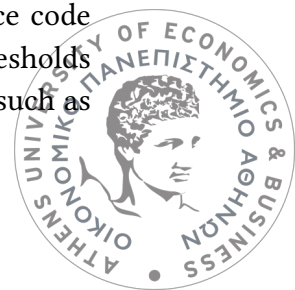
A complete list of the selected C# repositories along with their analyzed results can be found online [Sha19c]. Table 4.1 presents some key characteristics of the selected subject systems.

Table 4.1: Characteristics of the Analyzed Repositories

Attributes	Total values
Repositories	3 209
Components	75 205
Types	724 854
Methods	3 739 387
Lines of code (C# only)	83 135 679

4.1.2 Analyzing C# Repositories Using Designite

Designite [Sha16, SMT16] is a software design quality assessment tool. We use Designite's version 2.3.0 to analyze repositories. Apart from supporting detection of a wide variety of design and implementation smells, it detects seven well-known architecture smells for C# code. Other key features supported by the tool are object-oriented code metrics computation, dependency structure matrix, trend analysis of smells, code-clone detection, integration with external tools via its console application, and hotspot analysis. Apart from its GUI-based desktop application, Designite also offers a console application which is particularly useful for analyzing a large number of repositories automatically. Customization is one of the major features of the tool – a user can customize the way input source code is provided to the tool, certain smells to skip in an analysis session, or change thresholds that are used to detect certain smells. The tool provides interactive visualizations (such as



sunburst) for the detected smells and metrics; these visualization aids make it easier for the users to comprehend the results. Figure 4.1 provides an overview of major features of the tool. The tool offers free *academic* licenses for all academic purposes.

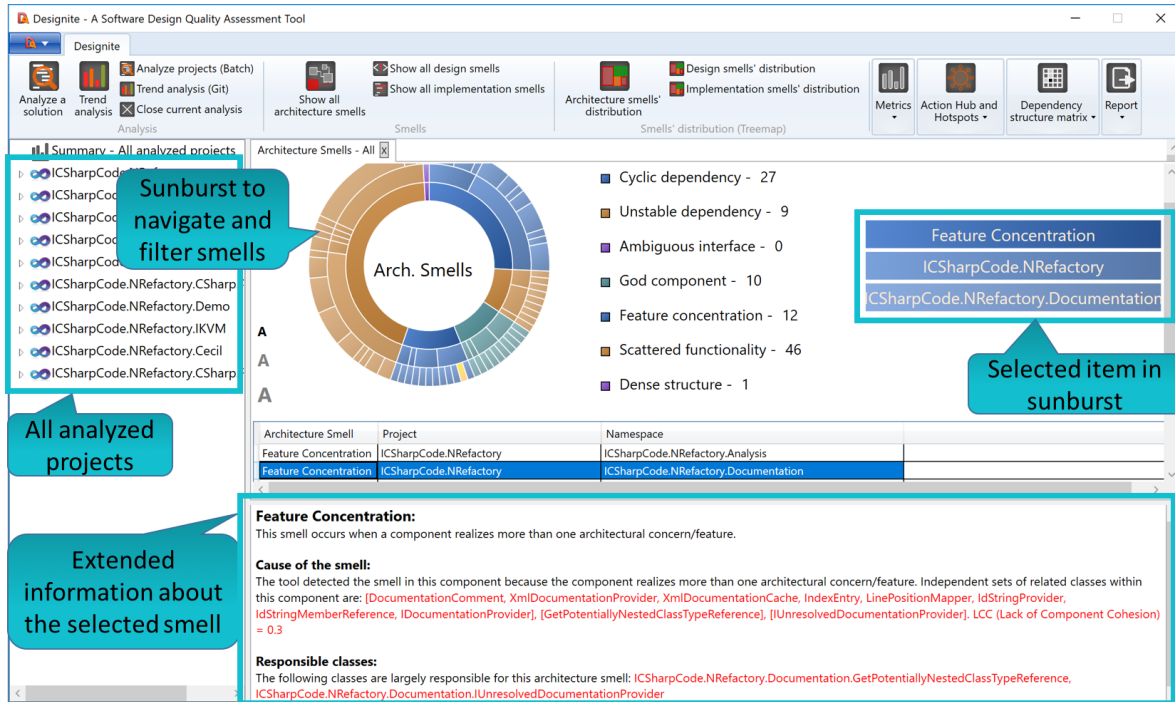


Figure 4.1: Presentation of identified smells in Designite

4.1.2.1 Architecture

Figure 4.2 shows the major components of the tool. Designite uses Roslyn¹ to parse C# code and prepares Abstract Syntax Tree (AST). The source model layer accesses the AST and prepares a simple hierarchical source code model. The model contains information about all the analyzed source code elements (such as namespaces, classes, methods, and fields). It is a hierarchical model; therefore, for example, an object of a project holds references to all the namespace objects in the project. The model is used by the tool's back-end to infer smells and compute metrics. The back-end hosts the domain logic *i.e.*, rules to detect smells. Apart from a desktop application, the tool offers Microsoft Visual Studio extension² as well as a console application.

4.1.2.2 Detection Mechanism for Supported Architecture Smells

In this section, we elaborate on the detection mechanism used to detect the supported architecture smells.

¹<https://github.com/dotnet/roslyn>

²<https://marketplace.visualstudio.com/items?itemName=designite>.
Designite



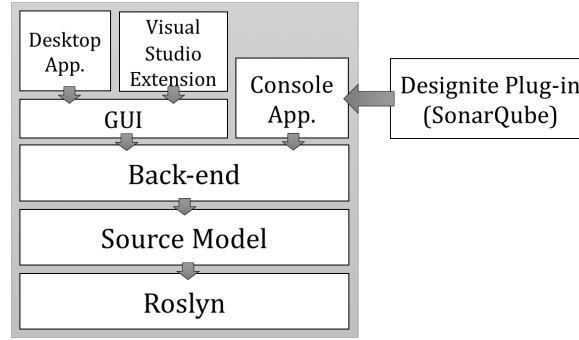


Figure 4.2: Architecture of the tool

Cyclic Dependency: To detect this smell, we first compute a dependency list for each component. Therefore, such a list for component A represents the components on which component A depends. Component A depends on component B if at least one of the classes in A refer (by association, aggregation, or composition) to at least one of the classes in component B. We construct a directed graph using the above information where ‘nodes’ refer to components and ‘edges’ refer to their dependencies. We then apply depth-first algorithm to detect cycles in the graph for each component. For large graphs, we stop the exploration after a threshold (currently set to 5 hops) to avoid extraneous computation.

Unstable Dependency: Instability of a component is computed as follows:

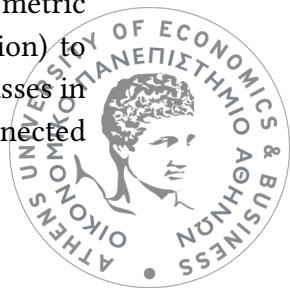
$$I = \frac{C_e}{C_e + C_a} \quad (4.1)$$

Here, I represents the degree of instability of the component, C_a represents the afferent coupling (or incoming dependencies), and C_e represents the efferent coupling (or outgoing dependencies). We compare the computed metric value of each component against its dependent components, and detect the smell when a dependent component is more stable.

Ambiguous Interface: We detect this smell when we find a component containing only one *public* or *internal* method. An *internal* method in C# has the visibility inside the assembly; hence, other components (namespaces) within the assembly may access it. In order to avoid small components from getting reported as ambiguous interfaces, we detect the smell only when the component has at least 5 classes.

God Component: We detect the smell when a component has more than 30 classes or 27,000 LOC following the recommendations by Lippert et al. [LR06].

Feature Concentration: Similar to LCOM (Lack of Cohesion of Methods) [CK94] metric which is applicable to classes, we compute LCC (Lack of Component Cohesion) to measure the cohesion of a component. To compute LCC, we identify related classes in a component, prepare a dependency graph, and identify the number of disconnected



sub-graphs. Two classes are related if they share any of the association, aggregation, composition, or inheritance relationships.

$$LCC = \frac{\text{Number of disconnected sub-graphs}}{\text{Total number of classes}} \quad (4.2)$$

This smell is detected if LCC is more than a pre-defined threshold. We use 0.2 as the LCC threshold to detect this smell.

Scattered Functionality: We determine the accesses to at least two external components that occur together from a method. If such accesses happen frequently (minimum 2 times) in a component, this indicates the presence of scattered functionality architecture smell.

Dense Structure: This smell occurs when components form a very dense dependency graph. In order to detect this smell, a dependency graph involving all the components is formed and the average degree of the graph is computed.

$$\text{Average degree} = \frac{2 \times |E|}{|V|} \quad (4.3)$$

Where E is the set of all the edges and V is the set of all vertices belonging to the graph. We detect the smell when the average degree is greater than a pre-defined threshold. We have set the threshold to 5. Since the dependency graph is formed by considering all the components present in the analyzed solution, maximum one instance of this smell can occur for the solution.

4.1.2.3 Detection Mechanism for Supported Design Smells

In this section, we present the detection method used to detect the supported design smells. Here, type/abstraction refers to a class or interface.

Duplicate Abstraction: We detect this smell when we find code clones (type-1) of size > 20 lines.

Imperative Abstraction: If a class has only one public method and the size of the class (in terms of LOC) is greater than a pre-defined threshold (*i.e.*, 100), we detect this smell.

Multifaceted Abstraction: We compute LCOM (Lack of Cohesion among Methods) metric for each type. If the value of the metric is greater than a threshold (*i.e.*, 0.8) and the type is not very small — number of fields and methods are greater than or equal to a threshold (*i.e.*, 7), we detect the smell.

Unnecessary Abstraction: If a type has no methods and the number of fields and properties are less than a threshold (*i.e.*, 5), we detect this smell.



Unutilized Abstraction: A type is unutilized if fan-in of the type is zero *i.e.*, there is no users of this type and if the type has no super class. In case the type has super class the the type suffers from unutilized abstraction if fan-in of both the type and its super class is zero.

Deficient Encapsulation: If a type has at least one public field or global field (declared as public static), we detect this smell.

Unexploited Encapsulation: We retrieve the list of types that are being explicitly checked in a method. We find the number of checked types that belong to the same inheritance hierarchy. If the number is greater than a threshold (*i.e.*, 2), we detect this smell.

Broken Modularization: If a type does not have any methods and count of fields and properties is greater than a certain threshold (*i.e.*, 5), we detect this smell.

Cyclically-dependent Modularization: We prepare a dependency graph of types from fan-in and fan-out information. We use this dependency graph to detect direct or indirect cycles.

Hub-like Modularization: If fan-out and fan-in of a type is greater than a threshold (*i.e.*, 20), the type suffers from this smell.

Insufficient Modularization: There are three forms of this smell.

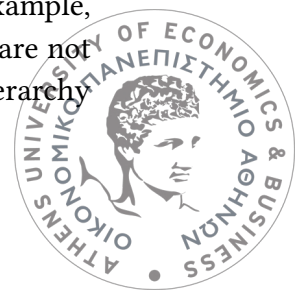
- If the count of public methods in a type crosses a threshold (*i.e.*, 20), the type is suffering from insufficient modularization.
- If the total methods in a type exceed a threshold (*i.e.*, 30), we detect this smell.
- We compute wmc (Weighted Methods per Class) metric for each type. We detect this smell if the value of the metric is more than a threshold (*i.e.*, 100).

Broken Hierarchy: For each class which has at least one super class with at least one public method, we check whether the class satisfy the condition of broken hierarchy smell. If the class does not have any method overridden or “leniently” overridden from its super classes, we detect the smell. A method is leniently overridden when the method name matches (but not necessarily the parameter types) with any of the public methods in the super classes.

Cyclic Hierarchy: If a type accesses any of the sub-types then we detect this smell.

Deep Hierarchy: We measure the DIT (Depth of Inheritance Tree) metric for each type. If the value of the metric crosses a threshold (*i.e.*, 6), we detect this smell.

Missing Hierarchy: We get a list of types checked explicitly in a method, for example, by using *instanceof* operator. Then, if more than one of the types in this list are not belonging to an inheritance hierarchy then we conclude that an inheritance hierarchy is missing.



Multipath Hierarchy: We derive a list of the direct super classes of a class. We also retrieve all the ancestors of all the parents. If there is any type in common between these two lists, we conclude the presence of this smell.

Rebellious Hierarchy: We check all the non-private methods in a class. If any method is overridden and either the method is empty or has only throw statement, then we detect this smell.

Unfactored Hierarchy: We detect this smell when we detect code clones in sibling types (where the classes share super type).

Wide Hierarchy: We compute the NC (Number of Children) metric for each type. If the value of the metric crosses a threshold (*i.e.*, 10), we detect this smell.

4.1.2.4 Detection Mechanism for Supported Implementation Smells

Complex Conditional: We detect this smell when a conditional expression (in statements such as *if*, *for*, and *while*) have more than three sub-expressions separated by logical operators.

Complex Method: We compute cyclomatic complexity [CK94] for all the methods. If cyclomatic complexity of a method crosses the threshold (*i.e.*, 8), the tool detects this smell.

Duplicate Code: When we find duplicate code blocks within a method, we detect this smell.

Empty Catch Block: We detect the smell when the try-catch statement has an empty catch block.

Long Identifier: We detect this smell when the length of an identifier (*i.e.*, local variable, parameter name, or a field) is greater than a threshold (*i.e.*, 30).

Long Method: When a method is longer than 100 LOC, we tag the method with this smell.

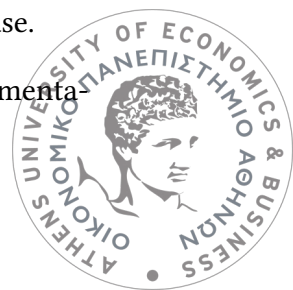
Long Parameter List: A method having more than five parameters, suffers from the long parameter list smell.

Long Statement: If the tool finds a statement larger (in terms of number of characters including white-spaces) than a threshold (*i.e.*, 120), it detects this smell.

Magic Number: The presence of a numeric literal (except 0 and 1) anywhere except assignment statements indicates this smell.

Missing Default: We detect this smell when *default* case is missing in a switch case.

Virtual Method Call from Constructor: This smell gets detected by our implementation, when a constructor calls at least one virtual method.



4.1.2.5 Evaluation

We conducted a manual validation to establish the accuracy of the tool. We chose a project *DotNetOpenAuth.Core* from a well-known open-source repository *DotNetOpenAuth*³ for the purpose. The selected project contains 16,663 LOC, 136 types, and 7 components. We sought help from two volunteers to carry out manual validation — one volunteer works in a software development company (three years of industry experience) and another volunteer is a PhD student with one year of industry experience. Both the volunteers did not work on the analyzed repository in advance; however, they have hands-on experience on working with complex industrial solutions and have a fair idea of software architecture and code smells.

We enforced the following protocol for the validation.

- Each volunteer carried out the initial manual analysis individually without discussing it with another volunteer.
- Given their industry experience, they were aware of the basic concept of smells and commonly known smells. Each volunteer picked all the considered design and architecture smells one by one and understood the semantics of the smell. We provided additional material to make their learning faster.
- Both the individuals went through all the source code files one by one patiently and checked the existence of each smell.
- While identifying smells, they were allowed to use IDE features such as *go to definition* and *list all references* as well as metrics generated from other tools.
- Once both the volunteers completed the analysis, we computed Cohen's Kappa [Coh60] to measure the mutual agreement between the volunteers' findings. We obtained $\kappa = 0.33$ as the value of Cohen's Kappa.
- Both the volunteers discussed their results, sorted out differences, and prepared a consolidated mutually agreed results. The consolidated results had 52 design and 18 architecture smells.
- At this point, they used Designite and analyzed the considered project and obtained a list of design and architecture smells.
- They compared the results obtained from the tool with their set of smells and tagged them as true-positive, false-positive, and false-negative. During this categorization, they observed that a subset of smells are identified by the tool which were not revealed by their manual analysis. They analyzed each of the smells in the subset and categorized them as well similar to the rest of the smells.

Table 4.2 presents the result of the manual validation showing smell instances detected by Designite and the consolidated set of smells identified by the volunteers. The table also

³<https://github.com/DotNetOpenAuth/DotNetOpenAuth>

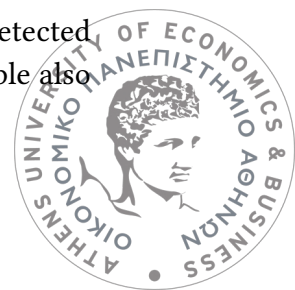


Table 4.2: Results of Manual Validation

Smells	Designite	Manual	FP	FN
Broken Hierarchy	2	1	1	0
Broken Modularization	2	3	0	1
Cyclically-dependent Modularization	34	34	0	0
Duplicate Abstraction	9	9	0	0
Hub-like Modularization	1	1	0	0
Imperative Abstraction	9	9	0	0
Insufficient Modularization	5	5	0	0
Multipath Hierarchy	1	1	0	0
Rebellious Hierarchy	2	2	0	0
Unnecessary Abstraction	20	19	1	0
Unutilized Abstraction	5	3	2	0
Wide Hierarchy	4	4	0	0
Cyclic Dependency	13	13	0	0
Unstable Dependency	4	4	0	0
God Component	1	1	0	0
Feature Concentration	5	5	0	0
Scattered Functionality	3	3	0	0
Dense Structure	1	1	0	0
	121	118	4	1

shows number of false-positives and false-negatives that we found in this validation. The detailed report showing individual smells along with the names of component or class where they occur and corresponding classification can be found online [Sha19d].

Interestingly, volunteers did not find all the legitimate smells manually; though, when the tool reported these instances, they found these instances true-positive. The highest number of smells that were missed by the volunteers are *cyclically-dependent modularization* and *cyclic dependency*. In this context, the volunteers found only unit-cycles *i.e.*, cycles involving only two classes or components. However, the tool reported cycles with more than one length. Volunteers verified all of these non-unit cycles and found them as true-positive. It implies that many smells go unnoticed even one actively looks for them; this observation emphasizes the importance of using tools.

The tool reported two false-positive instances of *unutilized abstraction*. Both of the instances are reported for exception types *i.e.*, classes that are used as custom exceptions. The tool could not resolve the instances of the types when they are thrown from a return statement.

The tool also fails to detect an instance of *broken modularization* smell. The volunteers classified the smell because the class only has a few data members and an empty constructor. However, due to the presence of the constructor, the tool did not identify the smell.

We compute precision and recall exhibit by the tool in the following way.

$$Precision = \frac{TP}{TP + FP}$$



$$Recall = \frac{TP}{TP + FN} \quad (4.5)$$

Here, TP, FP, and FN refer to true-positive, false-positive, and false-negative instances. Based on the above analysis, we obtain precision = $117/(117 + 4) = 96.6\%$ and recall = $117/(117 + 1) = 99.1\%$.

4.2 Detecting Smells using Deep Learning

In this section, we discuss the implementation details of the experiment in which we attempt to detect smells and explore the possibility of applying transfer-learning using deep learning methods. It includes data curation process starting from downloading repositories, detecting smells, generating positive and negative samples for training and evaluation of the models, and tokenizing samples. Also, we elaborate on the architecture of deep learning models.

4.2.1 Data Generation and Curation

In this section, we elaborate on the process of generating training and evaluation samples along with the tools used in the process. We download the C# and Java repositories and detect smells in the repositories using Designite. Designite results are used as ground truth for training as well evaluating the performance of the deep learning models. Further, we split each individual method or class and classify them into either a positive or negative sample based on the presence of the smell. Finally, we tokenize and preprocess each of the sample to feed them to deep learning models.

4.2.1.1 Downloading Repositories

We use the following protocol to identify and download our subject systems.

- We download repositories containing C# and Java code from GitHub. We use RepoReapers [MKCN17] to filter out low-quality repositories. RepoReapers analyzes GitHub repositories and provides scores for eight dimensions of their quality. These dimensions are architecture, community, continuous integration, documentation, history, license, issues, and unit tests.
- We select all the repositories where at least six out of eight and seven out of eight RepoReapers' dimensions have suitable scores for C# and Java repositories respectively. We consider a score suitable if it has a value greater than zero.
- We ensure that RepoReapers results do not include forked repositories.
- We discard repositories with fewer than five stars and less than 1,000 LOC.



- Following these criteria, we get a filtered list of 1,072 C# and 2,528 Java repositories. We select 100 repositories randomly from the filtered list of Java repositories. Finally, we download and analyze the 1,072 C# and 100 Java repositories.

4.2.1.2 Smell Detection

We use Designite to detect smells in C# code. Designite [SMT16, Sha16] is a software design quality assessment tool for code written in C#. It supports detection of eleven implementation, 19 design, and seven architecture smells. It also provides commonly used code metrics and other features such as trend analysis, code clone detection, and dependency structure matrix to help developers assess the software quality. A free academic license of Designite can be requested.

Similar to the C# version, we have developed DesigniteJava [Sha18c], which is an open-source tool for analyzing and detecting smells in a Java codebase. The tool supports detection of 17 design and ten implementation smells.

We use the console version of Designite (version 2.5.10) and DesigniteJava (version 1.1.0) to analyze C# and Java code respectively and detect the specified design and implementation smells in each of the downloaded repositories.

4.2.1.3 Splitting Code Fragments

CodeSplit is a set of two utility programs, one for each programming language, that split methods or classes written in C# and Java source code into individual files. Hence, given a C# or Java project, the utilities can parse the code correctly (using Roslyn for C# and Eclipse JDT for Java), and emit the individual method or class fragments into separate files following hierarchical structure (*i.e.*, namespaces/packages becomes folders). CodeSplit for Java is an open-source project that can be found on GitHub [Sha19b]. CodeSplit for C# can be downloaded freely online [Sha19a].

4.2.1.4 Generating Training and Evaluation Data

The learning data generator requires information from two sources—a list of detected smells for each analyzed repository and a path to the folder where the code fragments corresponding to the repository are stored. The program takes a method (or class in case of design smells) at a time and checks whether the given smell has been detected in the method (or class) by Designite. If the method (or class) suffers from the smell, the program puts the code fragment into a “positive” folder corresponding to the smell otherwise into a “negative” folder.

4.2.1.5 Tokenizing Learning Data

Machine learning algorithms require the inputs to be given in a representation appropriate for extracting the features of interest, given the problem in hand. For a multitude of machine learning tasks it is a common practice to convert data into numerical representations



before feeding them to a machine learning algorithm. In the context of this study, we need to convert source code into vectors of numbers honoring the language keywords and other semantics. Tokenizer [Spi19] is an open-source tool that provides, among others, functionality for tokenizing source code elements into integers where different ranges of integers map to different types of elements in source code. Figure 4.3 shows a small C# method and corresponding tokens generated by Tokenizer. Currently, it supports six programming languages, including C# and Java.

```

public void InternalCallback(object state)
123 → { 40 → 2003 41
2002 → Callback(State); 59
474 → try 2005 2006 2007 2008
123 → { 46 → 40 44 46 41
2004 → timer.Change(Period, TimeSpan.Zero); 59
125 → } 40 → 2009
329 → catch (ObjectDisposedException) 41
123 → {} 125
125 → }

```

Figure 4.3: Tokens generated by Tokenizer for an example

4.2.1.6 Data Preparation

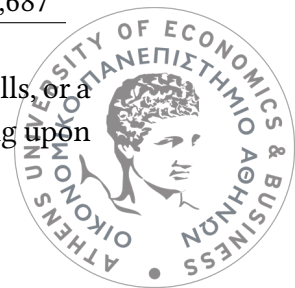
The stored samples are read into *numpy* arrays, preprocessed, and filtered. We first perform bare minimum preprocessing to clean the data—for both 1D and 2D samples, we scan all the samples for each smell and remove duplicates if any exist.

We split the samples in the ratio of 70-30 for training; *i.e.*, 70% of the samples are used for training a model while 30% samples are used for evaluation. We limit the maximum number of positive/negative training samples to 5,000. Therefore, for instance, if negative samples are more than 5,000, we drop the rest of the samples. We perform model training using balanced samples *i.e.*, we balance the number of samples for training by choosing the smaller number from the positive and negative sample count; we discard the remaining training samples from the larger side. Table 4.3 presents an example of data preparation.

Table 4.3: Number of samples in each step of preparing input data

		Initial samples	70-30 split	Applying max limit	Balancing
Positive	Training	4,961	3,472	3,472	3,472
	Evaluation		1,489	1,489	1,489
Negative	Training	175,623	122,936	5,000	3,472
	Evaluation		52,687	52,687	52,687

Each individual input instance, either a method in the case of implementation smells, or a class in the case of design smells, is stored in the appropriate data structure depending upon



the model that will use it. In 1D representation, each individual input instance is represented by a flat 1D array of sequences of tokens, compatible for use with the RNN and the CNN-1D models. In the 2D representation, each input instance is represented by a 2D array of tokens, preserving the original statement-by-statement delineation of source code thus providing the grid-like input format that is required by CNN-2D models. All the individual samples are stored in a few files (where each file size is approximately 50 MB) to optimize the I/O operations due to a large number of files. We read all the samples into a *numpy* array and we filter out the outliers. In particular, we compute the mean input size and discard all the samples with length over one standard deviation away from the mean. This filtering helps us keep the training set in reasonable bounds and avoids waste of memory and processing resources. We pad the input array with zeros to the extent of the longest remaining input in order to create vectors of uniform length and bring the data in the appropriate format for using with the deep learning models. Finally, we shuffle the array of input samples along with its corresponding labels array.

4.2.2 Architecture of Deep Learning Models

In this section, we present the architecture of the neural network models that we use in this study. The Python implementation of the experiment using the Keras library can be found online.⁴

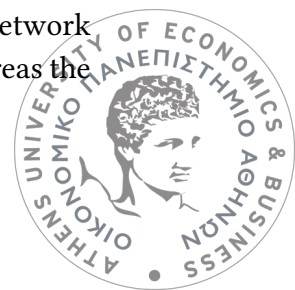
4.2.2.1 CNN Model

Figure 4.4 presents the architecture of CNN model used to detect smells. This architecture is inspired by typical CNN architectures used in image classification [KSH12] and consists of a feature extraction part followed by a classification part. The feature extraction part is composed of an ensemble of layers, specifically, convolution, batch normalization, and max pooling layers. This set of layers form the hidden layers of the architecture. The convolution layer performs convolution operations based on the specified filter and kernel parameters and computes accordingly the network weights to the next layer, whereas the max pooling layer effectuates a reduction on the dimensionality of the feature space. Batch normalization [IS15] mitigates the effects of varied input distributions for each training mini-batch, thus optimizing training. In order to experiment with different configurations, we use one, two, and three hidden layers.

The output of the last max pooling layer is connected to a dropout layer. Dropout performs another type of regularization by ignoring some randomly selected nodes during training in order to prevent over-fitting [SHK⁺14]. In our experiments we set the dropout rate for the layer to be equal to 0.1 which means that the nodes to be ignored are randomly selected with probability 0.1.

The output of the last dropout layer is fed into a densely connected classifier network that consists of a stack of two dense layers. These classifiers process 1D vectors, whereas the

⁴<https://github.com/tushartushar/DeepLearningSmells>



incoming output from the last hidden layer is a 3D tensor (that corresponds to height and width of an input sample, and channel; in this case, the number of channels is one). For this reason, a flatten layer is used first, to transform the data in the appropriate format before feeding them to the first dense layer with number of units = 32 units and *relu* activation. This is followed by the second dense layer with one unit and *sigmoid* activation. This last dense layer comprises the output layer and contains a single neuron in order to make predictions on whether a given instance belongs to the positive or negative class in terms of the smell under investigation. The layer uses the sigmoid activation function in order to produce a probability within the range of 0 to 1.

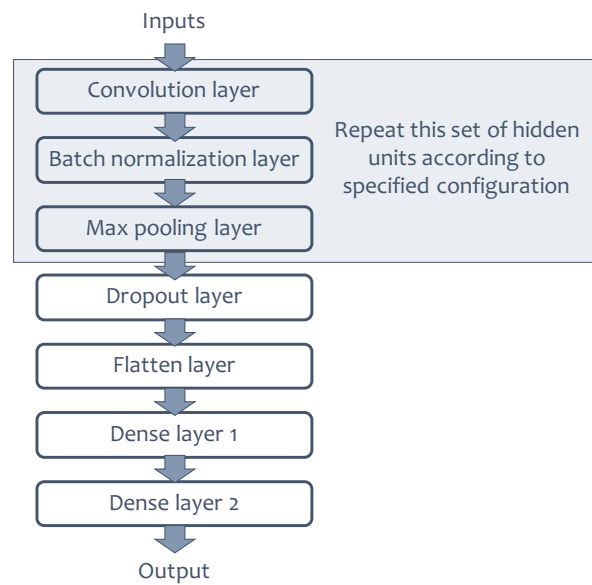


Figure 4.4: Architecture of employed CNN

We use dynamic batch size depending upon the size of samples to train. We divide the training sample size by 512 and use the result as the index to choose one of the items in the possible batch size array (32, 64, 128, 256). For instance, we use 32 as batch size when the training sample size is 500 and 256 when the training sample size is 2000.

The hyper-parameters are set to different values in order to experiment with different configurations of the model. Table 4.4 lists all the different values chosen for the hyper-parameters. We execute CNN models for 144 configurations that result from generating combinations of different values of hyper-parameters and number of repetitions of the set of hidden units. We label each configuration between 1 and 144 where configuration 1 refers to number of repetitions of the set of hidden units = 1, number of filters = 8, kernel size = 5, and pooling window size = 2. Similarly, configuration 144 refers to number of repetitions of the set of hidden units = 3, number of filters = 64, kernel size = 11, and pooling window size = 5. Both the 1D and 2D variants use the same architecture replacing the 2D version of Keras layers for their 1D counterparts.

We ensure the best attainable performance and avoid over-fitting by using *early stop*-

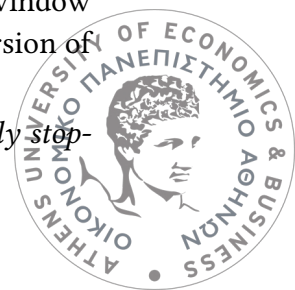


Table 4.4: Chosen values of hyper-parameters for the CNN model

Hyper-parameter	Values
Filters in convolution layer	{8, 16, 32, 64}
Kernel size in convolution layer	{5, 7, 11}
Pooling window size in max pooling layer	{2, 3, 4, 5}
Maximum epochs	50

*ping*⁵ as a regularization method. It implies that the model may reach a maximum of 50 epochs during training. However, if there is no improvement in the validation loss of the trained model for five consecutive epochs (since patience, a parameter to early stopping mechanism, is set to five), the training is interrupted. Along with it, we also use *model check point* to restore the best weights of the trained model.

For each experiment, we compute the following performance metrics — accuracy, ROC-AUC (Receiver Operating Curve-Area Under Curve), precision, recall, F1, and average precision score. We also record the actual epoch count where the models stopped training (due to early stopping). After we complete all the experiments with all the chosen hyper-parameters, we choose the best performing configuration and the corresponding number of epochs used by the experiment and retrain the model and record the final and best performance of the model.

4.2.2.2 RNN Model

Figure 4.5 presents the architecture of the employed RNN model which is inspired by state-of-the-art models in natural language modeling that employ an LSTM network as a recurrent layer [SSN12]. The model consists of an embedding layer followed by the feature learning part — a hidden LSTM layer. It is succeeded by the regularization (realized by a dropout layer) and classification (consisting of a dense layer) part.

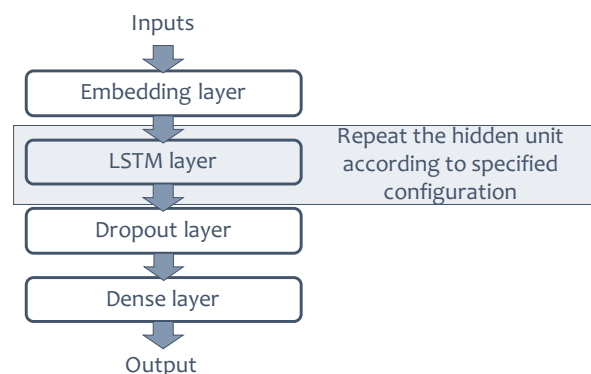


Figure 4.5: Architecture of employed RNN

The embedding layer maps discrete tokens into compact dense vector representations. One of the advantages of the LSTM networks is that they can effectively handle sequences of

⁵<https://keras.io/callbacks/>



varying lengths. To this end, in order to avoid the noise produced by the padded zeros in the input array, we set the *mask_zero* parameter to *True* provided by the Keras embedding layer implementation. Thus the padding is ignored and only the meaningful part of the input data is taken into account. We set *dropout* and *recurrent_dropout* parameters of LSTM layer to 0.1. The regular dropouts mask (or drop) network units at inputs and/or outputs whereas recurrent dropouts drop the connections between the recurrent units along with dropping units at inputs and/or outputs [GG15]. The output from the embedding layer is fed into the LSTM layer, which in turn outputs to the dropout layer. As in the case of the CNN model, we experiment for different depths of the RNN model by repeating multiple instances of the hidden layer.

The dropout layer uses a dropout rate equal to 0.2, which we empirically found effective for preventing over-training, yet conservative enough for avoiding under-training. The dense layer, which comprises the classification output layer, is configured with one unit and *sigmoid* activation as in the case of the CNN model. Similarly to the CNN model, we use *early stopping* (with maximum epochs = 50 and patience = 2) and *model check point* callbacks. Also, we use the dynamic batch size selection as explained in the previous subsection.

We try different values for the model hyper-parameters. Table 4.5 presents different values selected for each hyper-parameter. We measure the performance of the RNN model in 18 configurations by forming the combinations produced by the different chosen values of hyper-parameters and the number of repetitions of the set of hidden units.

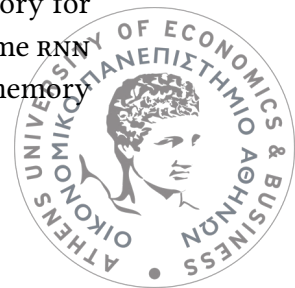
Table 4.5: Chosen values of hyper-parameters for the RNN model

Hyper-parameter	Values
Dimensionality of embedding layer	{16, 32}
LSTM units	{32, 64, 128}
Maximum epochs	50

As described earlier, we pick the best performing hyper-parameters and number of epochs and retrain the model to obtain the final and best performance of the model.

4.2.3 Hardware Specification

We perform all the experiments on the super-computing facility offered by GRNET (Greek Research and Technology Network). The experiments were run on GPU nodes (NVIDIA K40). Each GPU incorporates 2880 CUDA cores. We request 1 GPU node with 64 GB of memory for most of the experiments while submitting the job to the super computing facility. Some RNN experiments require more memory to perform the training; we request 128 GB of memory for them.



4.3 Analyzing Configuration Code for Quantitative Maintainability Assessment

We discuss the protocol that we used to select and download repositories containing Puppet code in this section. We also discuss the detection mechanism that we employ for detecting configuration smells using our tool Puppeteer.

4.3.1 Selecting and Downloading Puppet repositories

We follow the procedure given below to select and download the repositories.

We employ GHTorrent [Gou13, GS12] to select GitHub repositories to download. There are various options to choose from to select the subject systems such as number of commits and committers, stars, and number of relevant files in the repository. Each of the options (or their combinations) present different trade-offs. For instance, there are only 838 Puppet repositories that have five or more stars. We wanted to analyze larger number of repositories to increase the generalizability of the observations. Reducing the number of stars as a selection criterion would have resulted in more number of repositories; however, the significance of the criterion would have reduced. A high number of commits in a repository shows continuous evolution and thus we chose number of commits as the selection criterion. We choose to download all the repositories where the number of commits was more than or equal to 40. The above criterion provide us a list of 5,387 Puppet repositories to download. We download 4,621 repositories except some private ones.

Table 4.6 summarizes the characteristics of the downloaded repositories. We observed, by random sampling, that the downloaded repositories were either standalone Puppet-only repositories or system repositories (where production code as well as configuration code has been put together into a repository).

Table 4.6: Characteristics of the Downloaded Repositories

Attributes	Total count
Repositories	4,621
Puppet files	142,662
Class declarations	132,323
Define declarations	39,263
File resources	117,286
Package resources	49,841
Service resources	18,737
Exec declarations	43,468
Lines of code (Puppet only)	8,948,611

We analyze the downloaded repositories to detect implementation and design configuration smells. We use Puppet-Lint [Pup16c] tool to detect the majority of implementation configuration smells. We execute the tool on all the repositories and store the generated



output. In addition to using Puppet-Lint, we write our custom rules to detect the implementation configuration smells that the tool was not detecting (for instance, *complex expression* and *incomplete conditional*). We then aggregate the number of individual implementation smells that occur in each repository using the generated output and our mapping of best practices to the implementation smells (see Table 3.3).

We developed a tool, *Puppeteer* [Sha19e] to detect design configuration smells listed in Section 3.2.3.2. We discuss detection strategies of all the smells detected by Puppeteer in the rest of the section. The generated data by the tools mentioned above for both the smell categories for all the analyzed repositories can be found online [Sha].

4.3.2 Design Configuration Smells — Detection Strategies

This section discusses detection strategies that Puppeteer uses to identify design configuration smells.

Multifaceted Abstraction: The detection strategy for the two forms of the smell is as follows.

1. We compute a metric, '*Physical resources defined per resource declaration*', for each declared resource. We report the smell when the metric value is more than one.
2. We compute lack of cohesion for the configuration abstractions to detect the second form of the smell. In traditional software engineering, we use the LCOM (Lack of Cohesion Of Methods) [CK94] metric to compute lack of cohesion for an abstraction. The same metric cannot be used for configuration code due to its different structure and characteristics. We use the following algorithm to compute LCOM in a configuration code abstraction.
 - (a) Consider each declared element (such as file, package, service resources and exec statements) as a node in a graph. Initially, the graph contains the disconnected components (DC) equal to the number of elements.
 - (b) Identify the parameters of the abstraction, used variables, and literals (such as file name). Call them as data members collectively.
 - (c) For each data member, repeat the following: identify the components that uses the data member. Merge the identified components in a single component.
 - (d) Compute LCOM:

$$LCOM = \begin{cases} \frac{1}{|DC|} & \text{if } |DC| > 0 \\ 1 & \text{otherwise} \end{cases} \quad (4.6)$$

Note that we compute LCOM for each class, 'define', and file. Therefore, it is quite possible that the tool reports more than one instance of this smell in a single Puppet file.



Unnecessary Abstraction: We compute a metric namely ‘*Size of the abstraction body*’. A zero value of the metric shows that the abstraction does not contain any declarations and thus suffers from *unnecessary abstraction* smell.

Imperative Abstraction: We compute a metric namely ‘*Total ‘exec’ declarations*’ in a given abstraction. The tool reports the *imperative abstraction* smell when the abstraction has more than two ‘exec’ declarations and ratio of the ‘exec’ declarations against all the elements in the abstraction is more than 20%.

Missing Abstraction: We identify total number of configuration elements except classes or defines that are not encapsulated in a class or a ‘define’. A module suffers from the smell if there are more than two such elements in the module.

Insufficient Modularization: The detection strategy for the three forms of the smell is as follows.

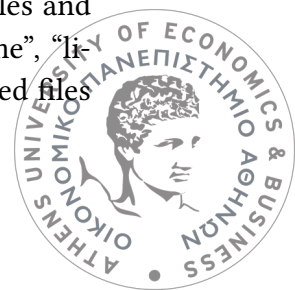
1. We count the number of classes and defines declared in a Puppet file. We report the smell if a file defines more than one class and ‘define’.
2. We count the number of lines in an abstraction. If a class or ‘define’ contains more than 40 lines of code, it suffers from the smell.
3. We compute maximum nesting depth for an abstraction. An abstraction with maximum nesting depth more than three suffers from this smell.

Duplicate Block: We use the PMD-CPD [CPD16] tool to identify code clones. A module suffers from this smell when a code clone of larger than 150 tokens gets identified in the module.

Broken Hierarchy: For all the class definitions, we identify the inherited class (if any). If the inherited class is defined in any other module, the class suffers from “broken hierarchy” smell.

Unstructured Module: The detection strategy for the three forms of the smell is as follows.

1. We search for a folder named “manifests” in the root folder of the repository. If the total number of Puppet files in the folder is more than five while there is no folder containing the string “modules”, the smell gets detected.
2. We find a folder containing the string “modules” and treat all the sub-folders as separate modules. Each module must have a folder named “manifests”. Absence of the folder shows the presence of the smell.
3. In each module, we count the unexpected files and folders. Expected files and folders are: “manifests”, “files”, “templates”, “lib”, “tests”, “spec”, “readme”, “license”, and “metadata”. A module with more than three such unexpected files or folders suffers from the smell.



Dense Structure: We prepare a graph for each repository to detect the smell. Each module is treated as a node and any reference from the module to another module is treated as an edge. We, then compute average degree of the graph.

$$AvgDegree(G) = \frac{2 \times |E|}{|V|} \quad (4.7)$$

where $|E|$ and $|V|$ are number of edges and nodes respectively. A graph with an average degree higher than 0.5 suffers from *Dense structure*: smell.

Deficient Encapsulation We count the number of global variables declared for each node declaration, followed by at least one include statement. If a node declaration has one or more such global variables, the module suffers from *deficient encapsulation* smell.

Weakened Modularity: We compute *modularity ratio* [BINF12] for each module as follows:

$$ModularityRatio(A) = \frac{Cohesion(A)}{Coupling(A)} \quad (4.8)$$

where, $Cohesion(A)$ refers to the number of intra-module references and $Coupling(A)$ refers to the number of inter-module references from module A . We report the smell if the ratio is less than one.

4.4 Analyzing Database Code for Maintainability Assessment

In this section, we discuss our method to select and mine repositories as well as the detection strategies that we employ in our tool *DbDeo*.

4.4.1 Mining Repositories

We used the following protocol to select the subject systems. We also illustrate the mechanism that we employ in extracting SQL statements and detecting smells.

4.4.1.1 Selecting Industrial Repositories

We approached two organizations SIG (Software Improvement Group) and SILO (Singular Logic) and sought access to their (or their clients') projects to analyze them. We analyzed a total of 840 projects that belong to various domains including banking, CRM, and telecom.

4.4.1.2 Selecting Open-source Repositories

We employ RepoReapers [MKCN17] to select subject systems for the study. RepoReapers provides assessment about GitHub open-source repositories on eight dimensions (architecture, community, continuous integration, documentation, history, license, issues, and unit



tests) along with number of stars. We select all the 16,057 repositories that score greater than zero for eight or nine dimensions. We download these repositories one by one, looked for SQL statements in each repository, and discard the repositories that does not have any SQL statements.

4.4.1.3 Extracting SQL Statements

We use regular expressions to extract SQL statements from the acquired repositories in *DbDeo*. We implement a two-step process to extract SQL statements. In the first step, we use relaxed regular expressions optimized for speed and in the second step we use stringent regular expression optimized for correctness.

4.4.1.4 Analyzing and Detecting Smells

We find 357 industrial projects and 2,568 open-source projects that contained SQL statements. Then, we compute metrics such as the number of SELECT, CREATE TABLE, and INSERT statements as well as the number of files belonging to each programming language and corresponding total lines of code. Finally, we analyze all the SQL statements from all the repositories using our tool *DbDeo* to detect database schema smells. The raw data generated by the tool can be accessed online [Sha18a].

Table 4.7 shows some characteristics of the analyzed repositories. On average, industrial projects are 3.87 times bigger than open-source projects by LOC (average LOC for industrial and open-source projects are 617,617 and 159,328 respectively) and 5.05 times bigger by number of SQL statements (average number of SQL statements for industrial and open-source projects are 455 and 90 respectively). Although, CREATE TABLE statements are the major source of information to detect schema smells, many times other SQL statements are required to detect these smells. For example, we require CREATE TABLE, CREATE INDEX, and SELECT statements in a repository to detect *index abuse* smell. Therefore, we extract SELECT, INSERT, UPDATE, and CREATE INDEX statements also in addition to CREATE TABLE statements. We analyze 393,989 SQL statements from 2,925 repositories (on average ≈ 135 SQL statements per repository).

4.4.2 DbDeo and Detection Strategies for Database Smells

We developed *DbDeo* — an open-source database smell detection tool [Sha18b]. The tool has a meta-model generator component that uses the third-party library SQLParse⁶ to parse SQL statements and prepare a meta-model. The meta-model component defines abstractions such as *CreateTableStmt* and *TableColumn* and organizes them in a hierarchical structure. For instance, a *CreateTableStmt* object contains a list of *TableColumn* objects. These abstractions contain information about the parsed SQL statements. For example, one of the attributes belonging to *CreateTableStmt* is *totalColumnsInTable*. The smell detection module in turn uses the meta-model to detect database schema smells.

⁶<https://github.com/andialbrecht/sqlparse>

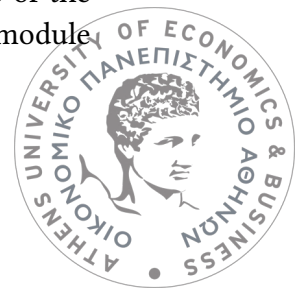


Table 4.7: Characteristics of the analyzed industrial (I) as well as open-source (OSS) repositories

Attributes	I	OSS
Initial set of repositories	840	16,057
Repositories with SQL statements	357	2,568
Files	2,559,984	3,297,932
Lines of code (source code only)	220,489,273	409,155,497
SELECT statements	51,652	74,096
CREATE TABLE statements	18,907	50,682
INSERT statements	74,416	66,830
UPDATE statements	10,454	29,002
CREATE INDEX statements	7,152	10,798

In the rest of this section, we discuss detection strategies employed by *DbDeo* to detect database smells.

Compound attribute: We look for pattern-matching expressions in an SQL query. In a SELECT statement, we check the presence of REGEX in a WHERE clause. We inquire whether a comma is used to separate values that are inserted against an attribute using an INSERT statement. For UPDATE statements, we check the use of a comma in the SET clause.

Adjacency list: We look for a foreign key constraint referring to an attribute in the same table.

Metadata as data: We look for a schema definition containing only three attributes. We detect the smell if we find two of the attributes, among three, of type VARCHAR.

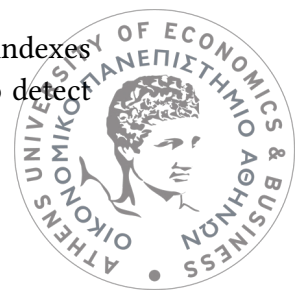
Multi-column attribute: We check the schema for a pattern '*<attribute>*'N where N is a number. We detect this smell in the table, if the schema has more than one attribute that matches with the above pattern.

Clone tables: We check all the schema definitions within a database for a pattern '*<Table name>*'N where N is a number. We conclude that a database has this smell when the database has two or more tables matching with the above pattern.

Values in attribute definition: We detect the smell by checking the schema for “enum” or “check” where the construct imposes a restriction on the possible values that can be entered for an attribute.

Index abuse: *Missing indexes* — We identify this variant of the smell when there exists at least one table and the number of indexes in the database are zero.

Insufficient indexes — commonly available database vendors support creating indexes for primary keys implicitly. We look for missing indexes for foreign keys to detect this smell variant.



Unused indexes — We identify this variant when the indexed attributes don't appear in any query.

God table: We count the total number of attributes defined in a schema definition. The table suffers from this smell if the number of attributes defined in the table crosses a threshold (currently we use 10 attributes as a threshold).

Overloaded attribute names: We scan all the attributes and their properties in schema definitions. If we find two or more attributes that have an identical name but defined as different data types, we report this smell.

We also considered detecting the remaining four smells automatically. However, we found it technically challenging to detect them automatically with high accuracy. For instance, *superfluous key* can be detected automatically if we have both the database schema and the data. However, devising heuristics without looking into data is prone to high false-positives.

4.4.3 Accuracy of DbDeo

We selected ten repositories randomly, performed each step listed in Section 4.4.1 (i.e., extract SQL code, compute basic metrics, and detect smells) on these repositories, and analyzed the output of each step.

4.4.3.1 Accuracy of the SQL Statements Extraction

An SQL statement may appear in host source code either independently (in separate files) or embedded in the host source code. Majority of the times, an embedded SQL statement receives some or all arguments dynamically by the host code. This property, along with diverse vendor-specific syntax of SQL statements, makes it difficult to cover all forms of SQL statements and extract them accurately using regular expressions. Brink *et al.* [BLV07] also reveals challenges in separating embedded SQL statements from host source code considering possible variations in host programming language and vendor specific SQL syntaxes. Given the importance of the extracted SQL statements' quality and associated challenges, we first assess the quality of the extracted SQL statements.

As mentioned earlier, *DbDeo* extracts SQL statements in two steps. In the first step, it extracts the SQL statements embedded in the source code using generic regular expressions. The tool employs a few heuristics and stringent regular expressions in the second step. The second step is rigorous and relatively more time consuming. Extracting potential SQL statements in the first step and then cleanse them gives us performance without compromising on the quality of the extracted statements.

We manually analyzed all the statements in the selected ten repositories and classified them either as an SQL statement, or as an incomplete SQL statement, an extraneous SQL statement, or a non-SQL statement. An extraneous SQL statement has valid SQL statement



followed by extraneous text or code that is not part of the SQL statement but was matched by the used regular expression.

Table 4.8 shows the performance of the SQL statement extraction process. We found two incomplete and two non-SQL statements in the extracted statements. One of the incomplete SQL statements is **CREATE TABLE xxx.yyy (...)**. Similarly, one of the non-SQL statements is **SELECT RANGE FROM ARCHIVE...** The statement is written as a comment but fulfils SQL grammar and thus gets extracted by the tool.

Table 4.8: Performance of the SQL extraction process

Total SQL statements	818
Incomplete SQL statements	2
Extraneous SQL statements	0
Non-SQL statements	2

4.4.3.2 Accuracy of Smell Detection

We detect database smells in all the ten repositories using *DbDeo*. We then verify each detected smell manually to measure the accuracy of the tool. Table 4.9 shows the total number of detected instances for each smell as well as the identified false-positive instances.

Table 4.9: Detected smells and identified false-positives

Smells	#Instances	Smells	#Instances
Compound attribute	4 (0)	Adjacency list	0 (0)
God table	26 (0)	Values in attribute definition	0 (0)
Metadata as data	3 (0)	Multicolumn attribute	15 (0)
Clone table	23 (0)	Overloaded attribute name	26 (2)
Index abuse	30 (0)		

As the table shows, we identified two false-positive instances in detected smells. The first false-positive instance of *overloaded attribute names* smell is found in the following CREATE TABLE statement (shown partially).

```

1 CREATE TABLE 'sql_nonce' (* 'id' INT UNSIGNED AUTO_INCREMENT NO NULL PRIMARY KEY,
2 * 'nonce' CHAR(64) NOT NULL,
3 ...

```

The tool detects the smell because the employed parser interprets “*” as the name of an attribute and the tool found another such attribute defined as different type in a different table. However, a manual inspection reveals that this SQL statement exists in a repository written mainly in C. The above SQL statement appears in a comment and the parser used in the tool does not differentiate comments from the rest of the code. Similarly, the source of another false-positive is also a misinterpretation by the parser. Apart from these instances, we find other detected instances as genuine cases of schema smells.



Chapter 5

Results and Discussion

The process of getting results is more important than the results itself.

In this chapter, we elaborate on the results that we obtained from our experiments and our documented observations. We present results from our maintainability analysis on C#, configuration, and database code as well as from our exploration on applying deep learning to detect code smells.

5.1 Results of Maintainability Analysis on Production Code

This section presents the results gathered from the maintainability analysis that we carried out for C# repositories and our observations *w.r.t.* each research question addressed.

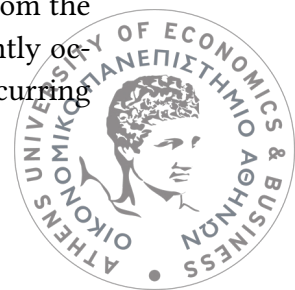
5.1.1 P-RQ1. What is the distribution of implementation, design, and architecture smells in C# code?

Approach

We compute the total number of detected smells for all the smells belonging to implementation, design, and architecture smell categories.

Results

Tables 5.1, 5.2, and 5.3 list the total number of instances detected for each smell. From the implementation smells side, *magic number* and *long statement* are the most frequently occurring smells. On the other hand, *virtual method call from constructor* is the least occurring implementation smell.





We observe that analyzed C# code on average contains one magic number per 16 lines of code. It is surprising to see a large number of *magic number* smells despite the fact that Designite excludes literals 0 and 1 while detecting the smell.

At design granularity, *cyclic-dependent modularization* and *unutilized abstraction* are the most frequently occurring smells. On the other hand, *Deep hierarchy* is the least occurring design smell.

Interestingly, *duplicate abstraction* is one of the most frequently occurring design smells but *duplicate code* is one of the least frequently occurring implementation smells. It is because the scope of the two smells differs significantly; clones belonging to *duplicate abstraction* occur anywhere in a project (but not in the same method) while clones belonging to *duplicate code* only occur within a method.

Table 5.1: Distribution of Implementation Smells

Implementation smell	#Instances	Smell density
Complex Conditional	21,643	0.4389
Complex Method	95,244	1.9317
Duplicate Code	17,921	0.3634
Empty Catch Block	14,560	0.2953
Long Identifier	7,741	0.1570
Long Method	17,521	0.3553
Long Parameter List	79,899	1.6205
Long Statement	462,491	9.3805
Magic Number	2,993,353	60.7130
Missing Default	23,497	0.4765
Virtual Method Call from Constructor	4,545	0.0921

Table 5.3 lists the total number of architecture smells in all the analyzed repositories. The table reveals that the *cyclic dependency* is the most frequently occurring architecture smell followed by *feature concentration* smell. One potential reason for the *cyclic dependency* to occur in a high volume is the permutations of the cycles due to one dependency that is mainly responsible for introducing a cycle. For instance, assume we have three components *A*, *B*, and *C* with the following dependencies: *A* depends on *B*, *A* depends on *C*, *B* depends on *A*, and *C* depends on *B*. Now, since *B* depends on *A*, not only the tool will detect a cyclic dependency between component *A* and *B*, but also another cycle among *A*, *B*, and *C*.

The *dense structure* smell has been detected the least number of times among the detected smells. This can mainly be attributed to the fact that, by definition, the smell can be detected at most once in a repository while all other architecture smells can be spotted multiple times in a repository. The smell has been detected in only approximately 10% of the analyzed repositories. We observed that the median of LOC computed for all the analyzed repositories is 4,391.5 while it is 29,147.5 for the repositories where the smell has been detected. It clearly indicates that the smell is more prone to occur in large repositories. However, the

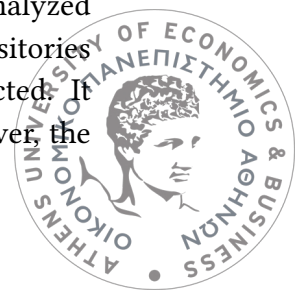


Table 5.2: Number of detected instances and smell density (per KLOC) of design smells in the analyzed repositories

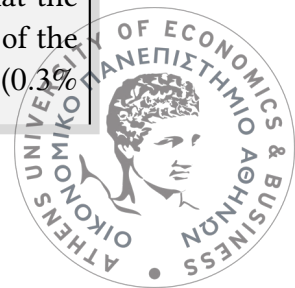
Design smell	#Instances	Smell density
Cyclically-dependent Modularization	193,188	2.3238
Unutilized Abstraction	182,638	2.1969
Duplicate Abstraction	118,429	1.4245
Unnecessary Abstraction	89,340	1.0746
Deficient Encapsulation	57,606	0.6929
Insufficient Modularization	35,595	0.4282
Broken Modularization	32,154	0.3868
Broken Hierarchy	29,668	0.3569
Unfactored Hierarchy	25,352	0.3049
Rebellious Hierarchy	23,371	0.2811
Imperative Abstraction	18,640	0.2242
Unexploited Encapsulation	11,299	0.1359
Cyclic Hierarchy	6,736	0.0810
Wide Hierarchy	4,838	0.0582
Multipath Hierarchy	3,359	0.0404
Missing Hierarchy	2,688	0.0323
Multifaceted Abstraction	2,104	0.0253
Hub-like Modularization	1,468	0.0177
Deep Hierarchy	286	0.0034

Table 5.3: Number of detected instances and smell density (per KLOC) of architecture smells in the analyzed repositories

Architecture smell	#Instances	Smell density
Cyclic Dependency	34 556	0.4157
Feature Concentration	17 420	0.2095
Scattered Functionality	11 623	0.1398
Unstable Dependency	10 195	0.1226
God Component	4 774	0.0574
Ambiguous Interface	852	0.0102
Dense Structure	302	0.0036

large size of a repository is not the only deciding factor. We find that 364 repositories are larger than the median LOC 29,147.5 where the smell does not occur. It implies that evolution of a software focused on quality may result in maintainable software systems.

Both *feature concentration* (at architecture granularity) and *multifaceted abstraction* (at design granularity) capture the cohesion aspect. It is surprising to note that the *feature concentration* smell occurs more often at architecture granularity (23% of the components) than its design granularity counterpart — *multifaceted abstraction* (0.3%).



of all the types). This clearly indicates that components are more prone to violate the single responsibility principle than the classes at design granularity. Therefore the software developers must pay attention to the component composition and cohesion when they extend the component.

5.1.2 P-RQ2. Do the detected smell instances belonging to different granularities correlate?

Approach

We compute the total instances of implementation, design, and architecture smells in each repository. We then compute the Spearman correlation coefficient between the detected instances of implementation and design smells. Similarly, we find the Spearman correlation between the sums of detected architecture and design smells. Further, we compute the Spearman coefficient between the individual pairs of architecture and design smells to observe the fine-grain correlation.

Results

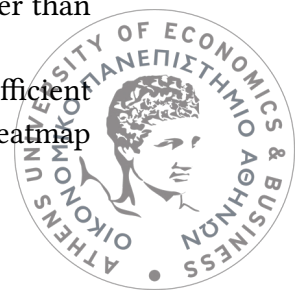
Figure 5.1a presents a scatter graph showing the co-occurrence between total instances of detected implementation and design smells. The Spearman correlation coefficient between implementation and design smell instances detected is 0.78059 (with $p\text{-value} < 2.2e - 16$).

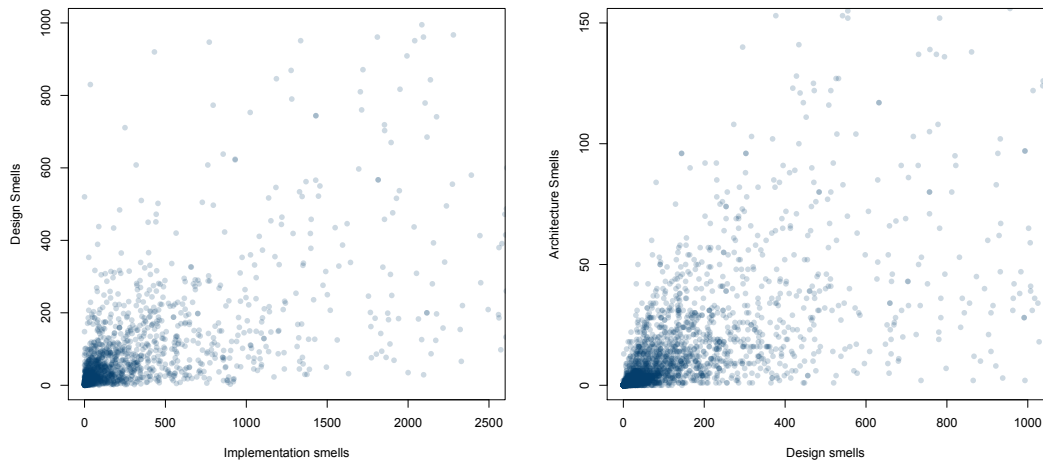
It shows that high volume of design (or implementation) smells is a strong indication of the presence of high volume of implementation (or design) smells in a C# project.

Similarly, the number of detected instances of architecture and design smells exhibit a high correlation coefficient (ρ) value of 0.86787 (with $p\text{-value} < 2.2e - 16$). Figure 5.1b shows a scatter plot between total detected instances of architecture and design smells in each repository. **This indicates that architecture smells exhibit very strong positive correlation with design smells.** Therefore, it can be inferred that a large population of design smell instances present in a repository indicates the presence of a high number of architecture smell instances and vice-versa. These observations might encourage a software developer to find and refactor architecture smells when she finds a large number of design smells in her software system.

To find deeper and more fine-grained relationships, we compute Spearman correlation coefficients between individual types of design smells and architecture smells. Figure 5.2 presents Spearman correlation coefficient values for all 133 architecture-design smell pairs in a heatmap. The darker color of a cell in the heatmap shows stronger correlation. A cell with coefficient in the red color shows statistical insignificant values ($p\text{-value}$ greater than or equal to 0.005).

The heatmaps in figure 5.2 show both the Spearman and Pearson correlation coefficient values for all 133 architecture-design smell pairs. The darker color of a cell in the heatmap





(a) Co-occurrence between detected implementation and design smells (b) Co-occurrence between detected design and architecture smells

Figure 5.1: Scatter plots showing co-occurrence between smells in two granularities

shows stronger correlation. A cell with coefficient in the red color shows statistically insignificant values (p-value greater than or equal to 0.005).

The correlation coefficient values show almost no correlation between individual architecture and design smells. The highest correlation shown by the smell pair *unused abstraction* and *feature concentration* ($\rho = 0.27$). Low correlation results are inline with the results presented by Macia *et al.* [MGP⁺12] where they found that 60% of the automatically detected code smells are not correlated with architecture smells.

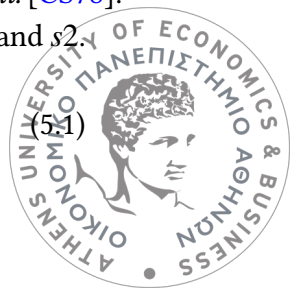
✍ The correlation analysis presented in this research question reveals that cumulatively design and architecture smells show a very strong correlation. However, fine-grain correlation analysis suggests that both the kinds of smells are not correlated and do not follow a monotonic relationship.

5.1.3 P-RQ3. Is the principle of coexistence applicable to smells in C# projects?

Approach

We compute the average intra-category co-occurrence for each smell. Co-occurrence is commonly used in bio-geography; we use the co-occurrence index used by Connor *et al.* [CS78]. The following equation computes the co-occurrence coefficient C between smells $s1$ and $s2$.

$$C(s1, s2) = \frac{n1 \times n2}{N}$$



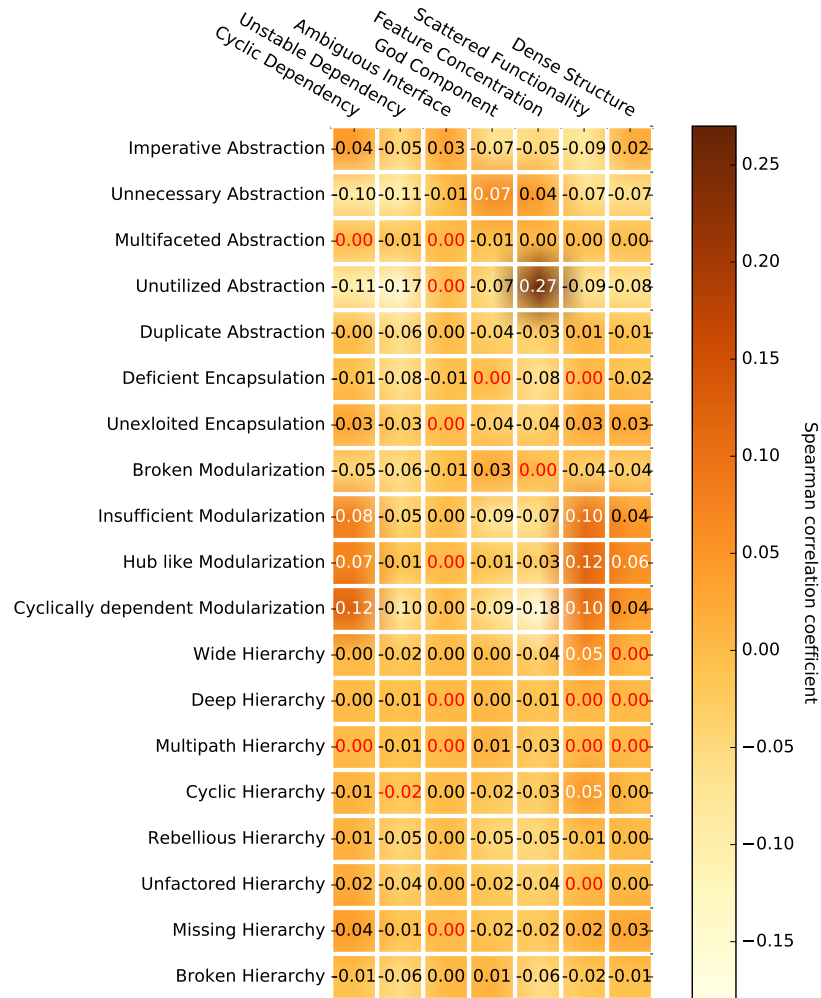


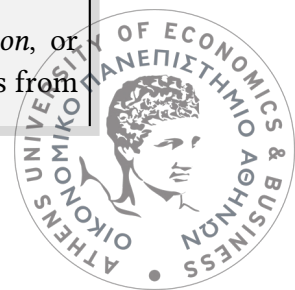
Figure 5.2: Correlation between individual architecture and design smells

Here, $n1$ and $n2$ are the number of detected instances of smells $s1$ and $s2$ respectively. N is the total number of detected smells in the repository.

Results

Figures 5.3, 5.4, and 5.5 present the average co-occurrence for each smell for all the three smell categories. *Cyclic dependency* shows the highest and *dense structure* shows the lowest co-occurrence in the architecture smells category. For the design smells category, *cyclically-dependent modularization* and *deep hierarchy* exhibit the strongest and weakest intra-category co-occurrence. Similarly, figure 5.5 shows that *magic number* and *virtual method call from constructor* exhibit the highest and lowest co-occurrence respectively in the implementation smells category.

It implies that whenever *cyclic dependency*, *cyclically-dependent modularization*, or *magic number* smells are found in C# code, it is more likely to find other smells from



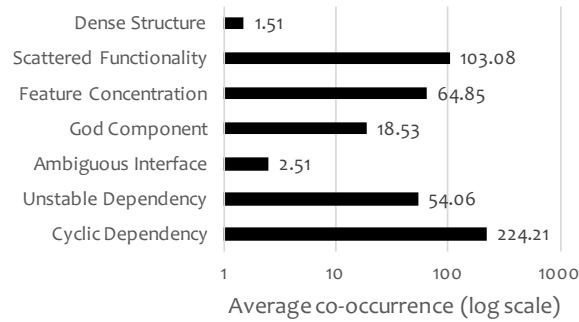


Figure 5.3: Average co-occurrence (intra-category) for architecture smells

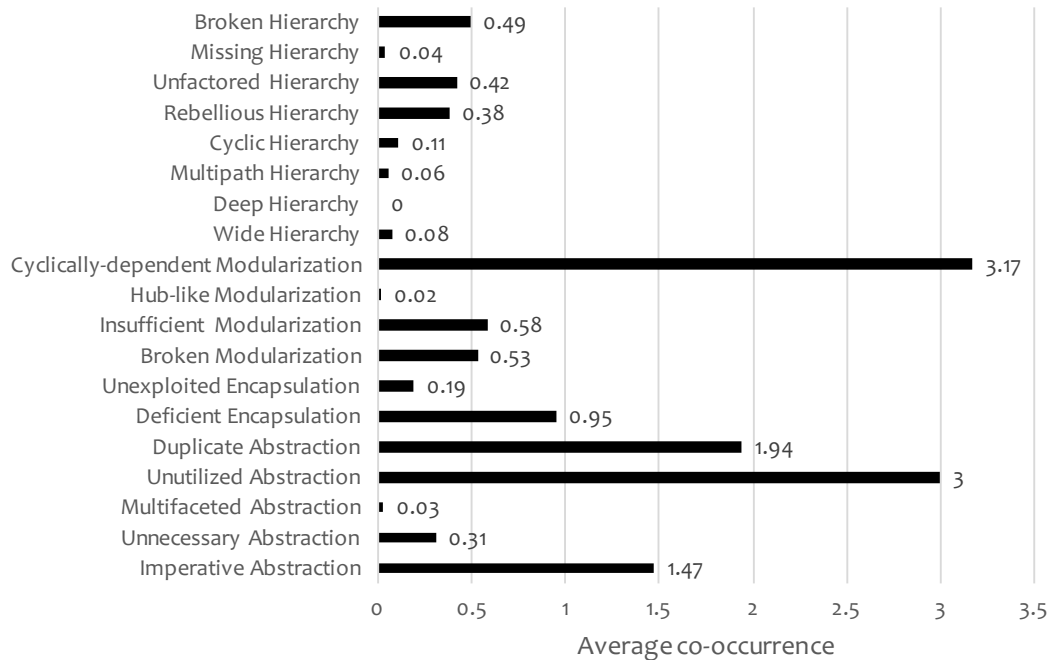


Figure 5.4: Average co-occurrence (intra-category) for design smells

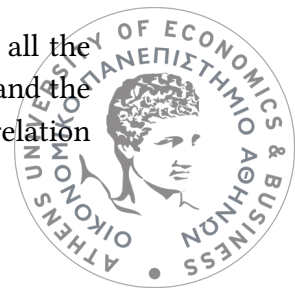
the same smell category in the project. On the other hand, the smells *dense structure*, *deep hierarchy* and *virtual method call from constructor* occur more independently.

Co-occurrence of implementation smells (figure 5.5) shows a large variation due to the huge difference in the number of detected instances for each smell in the category.

5.1.4 P-RQ4. Does smell density depend on the size of the C# repository?

Approach

We compute smell density for implementation, design, and architecture smells for all the analyzed repositories. We draw scatter plots between lines of code in a repository and the corresponding smell density for all the three smell categories. We also compute correlation



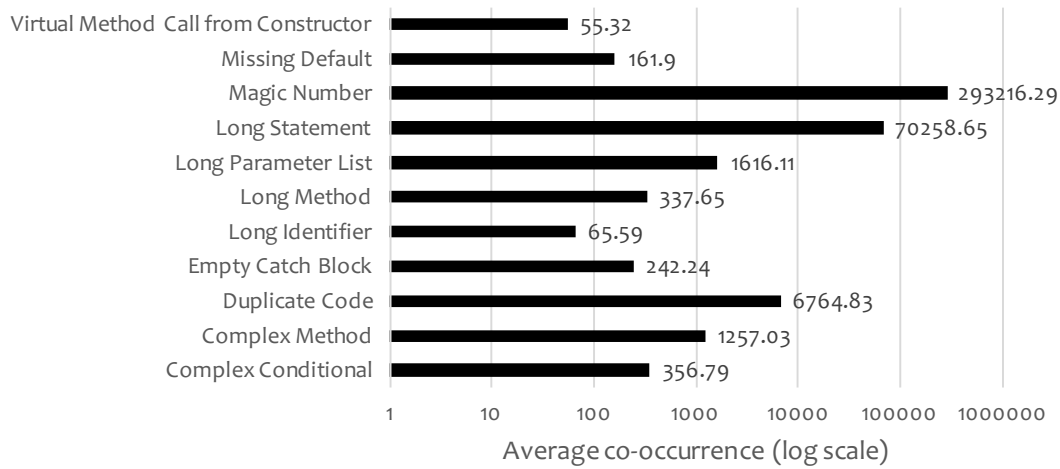


Figure 5.5: Average co-occurrence (intra-category) for implementation smells

coefficients for implementation, design, and architecture smell density and the repository size (in terms of LOC).

Results

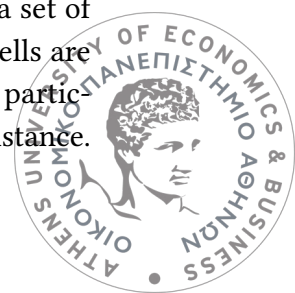
Figure 5.6 shows the distribution of smell density for implementation, design, and architecture smells against lines of code. A visual inspection of the above plots shows that the distribution of implementation as well as architecture smell density is more scattered and random than the distribution of design smell density. We compute the Spearman correlation coefficient between smell densities for all the three categories and LOC. The analysis reports 0.27800, -0.25426 , and 0.37476 as correlation coefficient ($p\text{-value} < 2.2e - 16$) w.r.t. implementation, design, and architecture smell density respectively. The results show a weak positive correlation for implementation smell density and weak negative correlation for design smell density with size of the project.

Given the low values for both the coefficients, the size of the project has low impact on the number of design and implementation smells. Architecture smell density exhibits moderate positive correlation indicating that the number of architecture smells increases as the size of the project grows.

5.1.5 P-RQ5. Are architecture smells collocated with design smells?

Approach

The architecture and design smells differ in granularity, hence they get reported in a set of components and a set of classes, respectively. To analyze whether both kinds of smells are collocated, we identify a set of *participating classes* for each architecture smell. A participating class contributes non-trivially to the occurrence of an architecture smell instance.



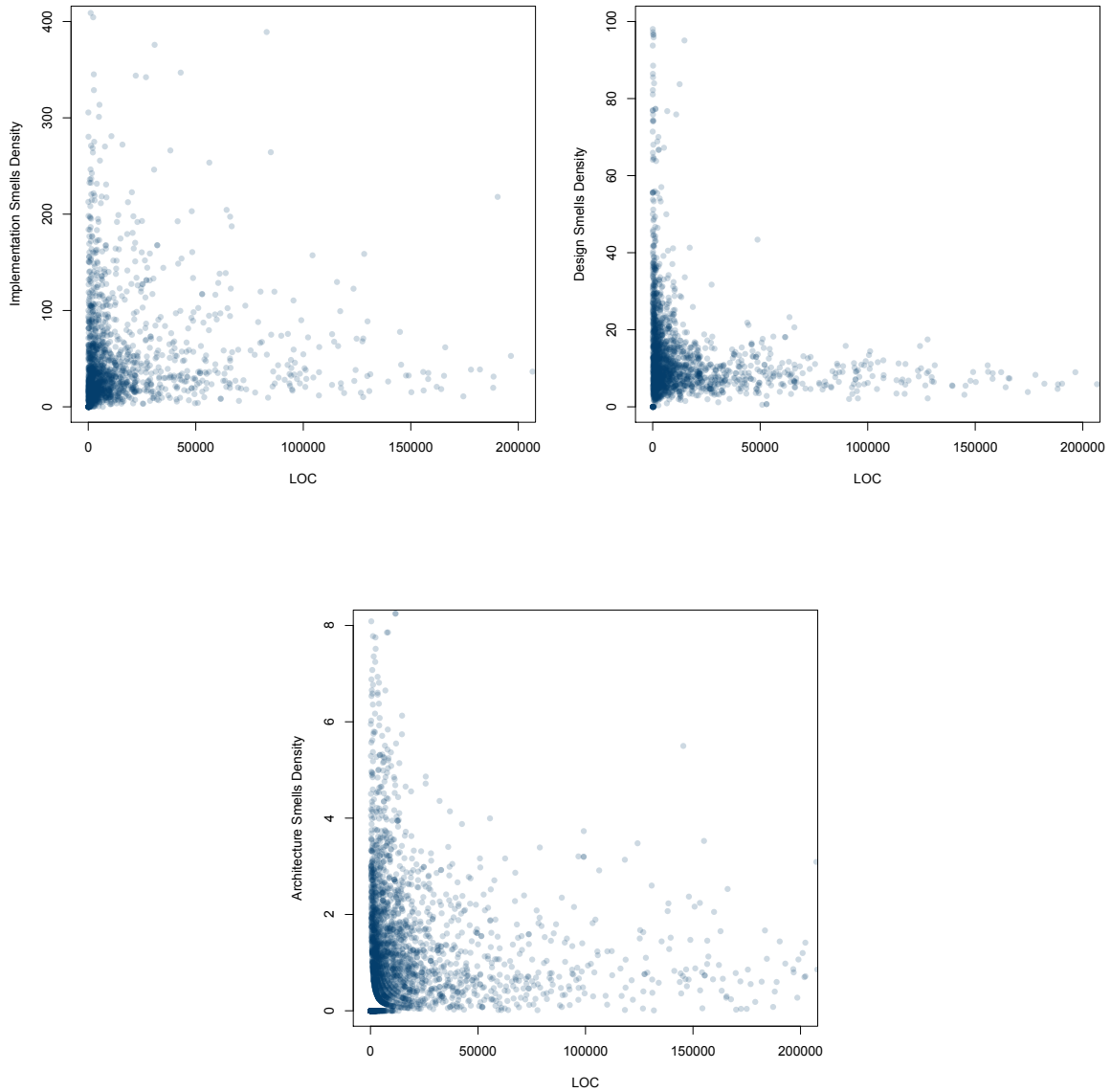
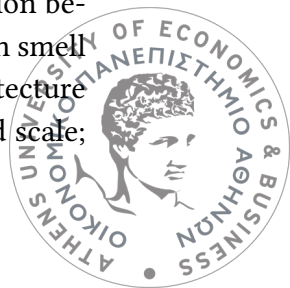


Figure 5.6: Smell density for implementation, design, and architecture smells against lines of code

Specifically, a design smell instance D and architecture smell instance A are considered to be “collocated” if a class reported by the instance D participates in the A instance.

We create a table containing all the classes belonging to all the analyzed repositories with their corresponding total architecture and design smell instances. We create 2×2 contingency matrices for both the smell categories and compute the phi-coefficient. Phi-coefficient, or mean square contingency coefficient, measures the degree of association between two binary variables. We perform the analysis for cumulative instances of both smell categories as well as 133 individual smell pairs. Naturally, the frequencies of architecture and design smell instances are not same due to the difference in the granularity and scale;



thus the instances of architecture smells are significantly lower than those of design smells. We have to normalize both numbers for semantically correct analysis and therefore we normalize the number of design smells by multiplying the ratio of the specific design and architecture smell.

Table 5.4: Contingency matrix for a design and architecture smell

		Design smell	
		1	0
Architecture smell	1	a	b
	0	c	d

Table 5.4 shows the contingency matrix for a design and architecture smell pair. The values of variables a , b , c , and d are used to compute phi-coefficient. However, as described above, we normalize the number of design smells instances (*i.e.*, c).

$$c' = c \times \frac{\text{Number of architecture smells}}{\text{Number of design smells}} \quad (5.2)$$

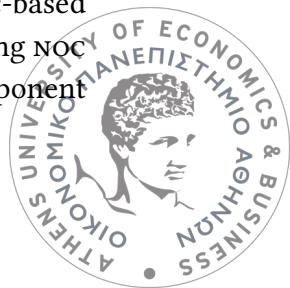
We compute the phi-coefficient using the following equation.

$$\phi = \frac{a \times d - c' \times b}{\sqrt{(a+b) \times (c'+d) \times (a+c') \times (b+d)}} \quad (5.3)$$

Mechanism to infer participating classes

To find out the collocation between architecture and design smell instances, we identify participating classes for each architecture smell. A participating class contributes to the architecture smell non-trivially. We formulated and implemented the following heuristics to infer participating classes for each architecture smell.

- *Cyclic dependency*: For each identified cycle, we find the classes (belonging to each component contributing to the formation of the cycle) that participate in the cycle. We include all these classes to the participating classes list.
- *Unstable dependency*: This smell occurs when a component depends on another component which is less stable than itself. In this case, all the classes that refer to classes of a less stable component are the participating classes for this smell.
- *Ambiguous interface*: We detect the smell when a component has only one public or internal method. We assign as the class responsible for the architecture smell the one that has the public or internal method.
- *God component*: The tool detects two variants of this smell. First, using LOC-based detection where the LOC of the component crosses a threshold and second, using NOC (Number of Classes)-based detection where the number of classes in the component crosses a threshold.



For the first smell variant, we sort the classes in a component by LOC in descending order. Then, we add classes from this list to the participating classes list one by one until the remaining size of the component becomes smaller than the threshold used for the smell detection. For the other variant, we choose the smallest classes (by number of methods) in the component assuming that these classes offer less functionality than the rest of the classes in the component. We select classes one by one in increasing order and add them to the list of participating classes until the remaining size of the component (in terms of number of classes) becomes smaller than the used threshold.

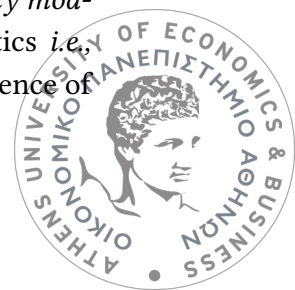
- *Feature concentration*: We detect the smell by inferring a dependency graph for each component. We find the size of disconnected sub-graphs within a component and sort them in ascending order. We add all the classes belonging to the sub-graphs to the participating classes list starting from the smallest sub-graph until the remaining sub-graphs show smaller LCC (Lack of Component Cohesion) than the selected threshold.
- *Scattered functionality*: In this smell, classes scattered in multiple components realize the same architectural concern. We identify these classes and tag them as participating for this architecture smell.
- *Dense structure*: We infer the dependency graph of the software, measure the degree of each component (which is the number of other components the component refers to), and sort them by decreasing degree. We add components one by one from this sorted list to the participating components until the cumulative degree of remaining components becomes smaller than the threshold used for detecting this smell. We include all the classes belonging to the identified responsible components that refer to classes belonging to other components.

Results

We found that cumulative collocation between architecture and design is low based on the computed phi-coefficient = 0.32. We also computed phi-coefficients for individual architecture and design smell pairs. Figure 5.7 shows collocation analysis heatmap of architecture and design smells. Each cell shows the computed phi-coefficient between an architecture-design smell pair.

The phi-coefficient between *feature concentration* and *unutilized abstraction* shows the highest collocation. This collocation makes sense because the presence of one or more unutilized abstractions increase the value of LCC for a component. This increased value of LCC in turn leads to feature concentration smell as discussed in detection mechanism for the smell (Section 4.1.2.2).

Along the expected lines, *cyclically-dependent modularization* design smell show relatively higher collocation with *cyclic dependency* architecture smell. The smell also collocates with *scattered functionality* and *dense structure* architecture smells. *Cyclic-dependency modularization* as well as both the architecture smells share the common characteristics i.e., coupling. It is understandable that presence of one of these smells indicates the presence of other related smells.



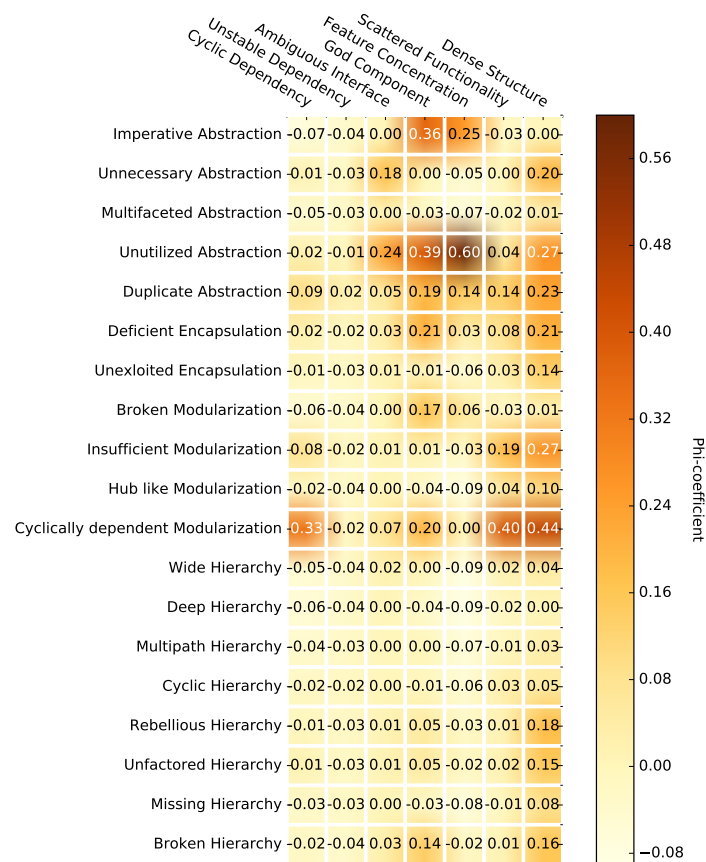


Figure 5.7: Collocation analysis between architecture and design smells



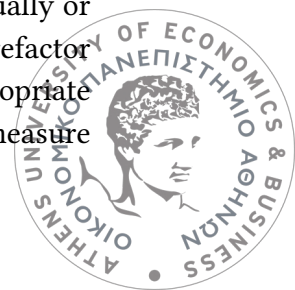
The majority of the collocation coefficients shows very low or no collocation among the smell pairs suggesting that the majority of the architecture and design smells do not collocate with each other.

5.1.6 P-RQ6. Can the refactoring of design smells lead to fewer architecture smells?

Approach

To further expand the analysis exploring the relationship between architecture and design smells, we examine the impact of design smell refactorings on architecture smells.

In order to perform this analysis, we need to obtain the refactored state of the project where all detected design smells are eliminated. It is prohibitively expensive to carry out refactoring for all detected design smells for a large number of projects either manually or automatically. There are numerous refactoring techniques that can be applied to refactor a smell [SSS14] based on the context; the present set of tools cannot adopt an appropriate refactoring automatically. Though it would be ideal to refactor all the projects to measure



the impact of the refactoring, given the above challenges it is not feasible. To overcome the challenge, we simulate the refactored state in which we assume that all the design smells are refactored. We propose a *theoretical basis* in which we map a set of design smells that may impact the existence of an architecture smell, derive a set of *consequences* of the assumed refactoring applied to remove each design smell instance, and *analyze* the code again to detect the architecture smell instance considering the corresponding discovered consequences.

We illustrate the mechanism described above with the help of an example for *cyclic dependency* architecture smell. Our theoretical base is that refactoring a *cyclic dependency modularization* design smell may lead to the removal of a *cyclic dependency* architecture smell. To find out whether the refactoring of a *cyclic dependency modularization* smell removes a *cyclic dependency* architecture smell, we first find out the consequences of refactoring an instance of a *cyclic dependency modularization* smell. The consequence of the refactoring is the removal of a dependency (say between X and Y classes) among all the classes that participate in the cycle. Then, we detect the *cyclic dependency* architecture smell again, this time considering that there is no dependency between classes X and Y. If the design smell refactoring removes the dependency that was causing the architecture smell, then the refactoring has an impact on the corresponding architecture smell.

We implement an extension of Designite to carry out the simulation. The extension processes the smell information generated by Designite, uses the mapping defined in this section, and computes the potential influences of design smell refactorings on architecture smells. We simulate the refactoring separately for each architecture smell. We compare the number of architecture smell instances before and after the refactoring.

Potential influences of design smell refactoring on architecture smells

We outline the theoretical basis in the form of potential influences of design smells refactoring corresponding to each architecture smell and mechanism to derive consequences of design smells refactoring in the subsection below.

- *Cyclic Dependency*: Refactoring a *cyclic-dependent modularization* design smell may influence the occurrences of a *cyclic dependency* architecture smell. We figure out the number of participating components in a *cyclic-dependent modularization* smell instance. If the number of participating components are more than one, then the design smell may impact *cyclic dependency* smell instances. To know whether the refactoring of an instance of a *cyclic-dependent modularization* removes any *cyclic dependency* smell, we follow the following heuristic. A *cyclic-dependent modularization* is removed when one of the dependencies in the cycle is removed. If classes A, B, and C form a cycle then the smell *cyclic-dependent modularization* will no longer exist if we remove a dependency (for instance $C \rightarrow A$). In this case, we need to check if the same class (say C) is present in the *cyclic dependency* architecture smell. If C is the only class from its component in the cycle and has only one dependency to one of the involved components, then the *cyclic dependency* architecture smell will be removed by refactoring the *cyclic-dependent modularization* smell.
- *Unstable dependency*: Refactoring *unutilized abstraction* may influence the occurrences



of this architecture smell. To know whether refactoring of an instance of *unutilized abstraction* removes the *unstable dependency* smell, we recompute the instability metric for each component after removing classes suffering from *unutilized abstraction* smells.

- *Ambiguous interface*: Occurrences of *ambiguous interface* may get influenced by refactoring the *unutilized abstraction* design smell. We check whether the class responsible for the architecture smell is unutilized to find whether refactoring an instance of *unutilized abstraction* removes the *unstable dependency* smell.
- *God component*: At first glance, one may think that refactoring *insufficient modularization* may influence the occurrence of *god component* architecture smell. However, refactoring *insufficient modularization* does not play a role in this context because when one refactors it, the refactoring may reduce the size of the class but one has to move the remaining functionality to another (possibly in a new) class. Hence, in most cases component size does not change and there is no influence on *god component* smell.

Carrying out refactoring for the *unutilized abstraction* design smell may impact the occurrences of *god component*. For the first (LOC-based) variant, we subtract the LOC associated with *unutilized abstraction* classes in the component and check the new LOC of the component against the corresponding threshold. For other (NOC — Number of Classes) variant, we reduce the number of classes suffering from *unutilized abstraction* and detect the smell again.

- *Feature concentration*: Refactoring *unutilized abstraction* may influence the detection of the *feature concentration* architecture smell. To find whether refactoring an instance of *unutilized abstraction* removes any instance of the *feature concentration* smell, we recompute PCC after removing the classes suffering from *unutilized abstraction*.
- *Scattered functionality*: Refactoring *unutilized abstraction* may seem to influence the detection of the *scattered functionality* architecture smell. However, if the class is not used then it cannot participate in this architecture smell. Therefore, no refactoring of any design smell is expected to influence the occurrence of this architecture smell.
- *Dense structure*: The occurrence of the *dense structure* architecture smell may get influenced by the refactoring of the *unutilized abstraction* design smell. To find the impact on the occurrence of a specific instance, we recompute the average degree of all the components after removing classes suffering from *unutilized abstraction* and check whether the *dense structure* smell still persists.

Results

We employ the extension of Designite that implement the mechanism described above to figure out potential influences of design smells refactoring on architecture smells. We perform



the analysis using the extension and detect architecture smells in the same set of repositories while taking the consequences of the design smells refactoring into consideration. Table 5.5 shows the number of architecture smells detected before and after the refactoring simulation for design smells. Figure 5.8 shows the percentage of architecture smells removed after the design smells refactoring.

Table 5.5: Architecture smell instances detected before and after the refactoring simulation for design smells

Architecture smell	#Instances (before)	#Instances (after)
Cyclic Dependency	34 556	29 573
Unstable Dependency	10 195	9 431
Ambiguous Interface	852	599
God Component	4 774	3 235
Feature Concentration	17 420	13 301
Scattered Functionality	11 623	11 623
Dense Structure	302	284

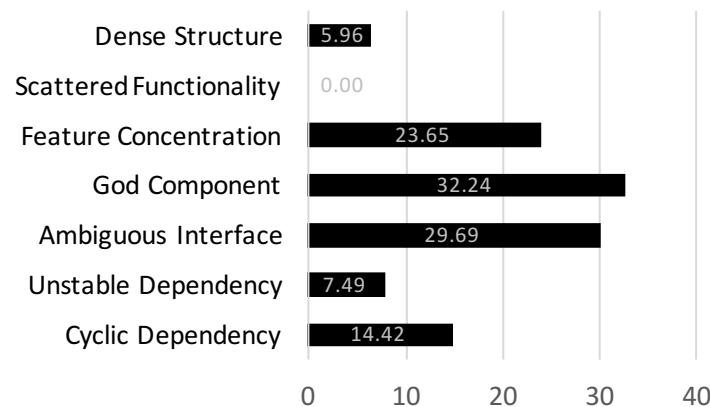


Figure 5.8: Removed architecture smells (in percentages) after simulating design smells refactoring

God component and *ambiguous interface* are the most influenced architecture smells — this indicates that refactoring design smells removes these two smells significantly. On the other hand, there is no impact of design smells refactoring on *scattered functionality* and there is a little impact on instances of *dense structure*.

The results of this exploration are inline with the collocation analysis. The collocation analysis shows high collocation for *feature concentration* (0.60), *god component* (0.39), and *ambiguous interface* (0.24) with *unutilized abstraction* design smell. Hence, refactoring *unutilized abstraction* lead to removing these three architecture smells as shown in the analysis presented in this section. Similarly, *cyclic dependency* and *dense structure* also exhibit high collocation with *cyclically-dependent modularization* (0.33) and *unutilized abstraction* (0.27).





With this exploration, we observe that refactoring smells at design granularity may remove smells at architecture granularity to a degree as high as one third of the smells (in case of *god component*). However, our exploration also leads to another important observation — refactoring smells is important at all granularities; design smells refactoring may remove some instances of architecture smells but a large number of architecture smells remain even after all the design smells are refactored.

5.1.7 Discussion and Implications

In this section, we extend our discussion on the smell relationships explored in this work. We also include implications for the software development community.

5.1.7.1 Discussion

We examine the collocation relationship between architecture and design smells; the results show that they exhibit selective collocation. It implies that though architecture smells arise from code and implementation choices made during the software development, their causal domain is larger and they have their individuality different from smells at design granularity.

We explore correlations between architecture and design smells cumulatively as well as between individual pairs. Very high correlations may indicate that a given smell is *superfluous*. For example, tracking humans' left-eye and right-eye colors will show an extremely high correlation between the two, and consequently storing only eye color is enough. Although, our analysis shows very strong correlation between the two kinds of smells when considered cumulatively, we observed almost no correlation for the individual smell pairs. This result demonstrates that each smell provides value-adding information. Furthermore, interestingly, even the similar smells at different granularities do not show strong correlation. Five architecture smells have corresponding similar smells at design granularity; it means that these smells represent and capture the same concept at different granularities. These smell pairs with their corresponding correlations are *cyclic dependency* — *cyclic-dependent modularization* ($\rho = 0.12$), *feature concentration* — *multifaceted abstraction* ($\rho = 0$), *scattered functionality* — *broken modularization* ($\rho = -0.04$), *god component* — *insufficient modularization* ($\rho = 0.09$), and *ambiguous interface* — *imperative abstraction* ($\rho = 0.03$). The almost no correlation outlines the non-monotonic relationship between these smell pairs and further emphasize the individuality of these smells.



Implications of Our Findings

We infer the following implications for the software development community.

Software development teams must detect, analyze, and refactor smells at all granularities. This implication is derived from our correlation analysis for smells arising at different granularities. Our results show that the presence of high volume of de-



sign smells indicates presence of high number of architecture smells and vice versa. Existing tools (such as NDepend^a and SonarQube^b mainly detect implementation and some design issues. Due to this limitation, a software development team using these tools perceives only a limited set of quality issues and thus issues at higher granularities go unnoticed. Furthermore, we observed that a significant amount of architecture smells persists even if all the detected design smells were refactored. This result also emphasizes the importance of detecting and refactoring smells at all granularities.

The software development community must avoid cycles among classes as well as among components to keep structure of the software easy to understand. Our results show that cyclic dependencies at both design and architecture granularities occur the most in open-source C# repositories. Higher number of cycles in a software introduce tangles and make the software difficult to comprehend.

Software development teams must pay more attention to their code quality as size of their software grows. This implication is derived from our analysis in which we find that architecture smell density tends to increase as the software grows. Actively used software systems grow; however, whether the software evolves by keeping the focus on code quality defines maintainability of the software. For example, in our analysis, *dense structure* smell has been detected in only approximately 10% of the analyzed repositories. We observed that the median of LOC computed for all the analyzed repositories is 4,391.5 while it is 29,147.5 for the repositories where the smell has been detected. It clearly indicates that the smell is more prone to occur in large repositories. However, the large size of a repository is not the only deciding factor. We figure out that 364 repositories are larger than the median LOC 29,147.5 where the smell does not occur. It implies that evolution of a software focused on quality may result in a maintainable software system.

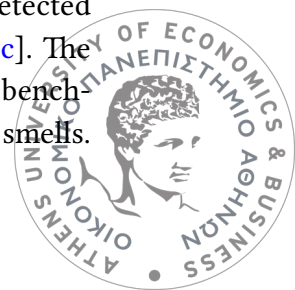
^a<http://www.ndepend.com/>

^b<https://www.sonarqube.org/>

5.1.7.2 Secondary Uses of this Work

We have added support to detect seven architecture smells in Designite. The software development community may use the tool to analyze their source code and improve maintainability of their code. The research community may utilize the tool to carry out studies concerning code smells. The tool is available online [Sha16] and free for all academic purposes.

Smells mining dataset is the basis of the research questions addressed in this work. The dataset contains all the supported implementation, design, and architecture smells detected in 3,209 open-source repositories. We have made the dataset available online [Sha19c]. The software engineering research community may utilize it in many ways including benchmarking and comparison as well as exploring other dimensions of source code with smells.



5.2 Results of Detecting Smells using Deep Learning

In this section, we elaborate on the results of our experiments exploring the application of deep learning methods for smell detection and present our observations.

5.2.1 D-RQ1. Is it possible to use deep learning methods to detect code smells? If yes, which deep learning method performs superior?

Approach

We prepare the input samples as described in Section 4.2.1. Table 5.6 presents the number of positive and negative samples used for each smell for training and evaluation; CNN-1D and RNN use 1D samples and CNN-2D uses 2D samples. As mentioned earlier, we train our models with the same number of positive and negative samples. Sample size for *multifaceted abstraction* (MA) is considerably low compared to other smells because each sample in this smell is a class (other smells use method fragments). The one-dimensional sample counts are different from their two-dimensional counterparts because we apply additional constraint for outlier exclusion, on permissible height, in addition to the width.

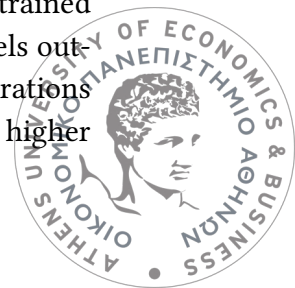
Table 5.6: Number of positive (P) and negative (N) samples used for training and evaluation for RQ1

	CNN-1D and RNN			CNN-2D		
	Training	Evaluation		Training	Evaluation	
	P and N	P	N	P and N	P	N
CM	3,472	1,489	51,926	2,641	1,132	45,204
ECB	1,200	515	52,900	982	422	45,915
MN	5,000	5,901	47,514	5,000	5,002	41,334
MA	290	125	22,727	284	122	17,362

Results

Figure 5.9 presents the performance (F1) of the models for the considered smells for all the configurations that we experimented with. The results from each model perspective show that performance of the models varies depending on the smell under analysis. Another observation from the trendlines shown in the plots is that performance of the convolution models remains more or less stable and unchanged for different configurations while RNN exhibits better performance as the complexity of the model increases except for *multifaceted abstraction* smell. It implies that the hyper-parameters that we experimented with do not play a very significant role for convolution models.

Figure 5.10 presents the boxplots comparing for each smell performance of all trained models, under all configurations. For *complex method* smell, both convolution models outperform the RNN. In between the convolution models, overall the various configurations of the CNN-1D model appear accumulated around the mean, whereas CNN-2D shows higher



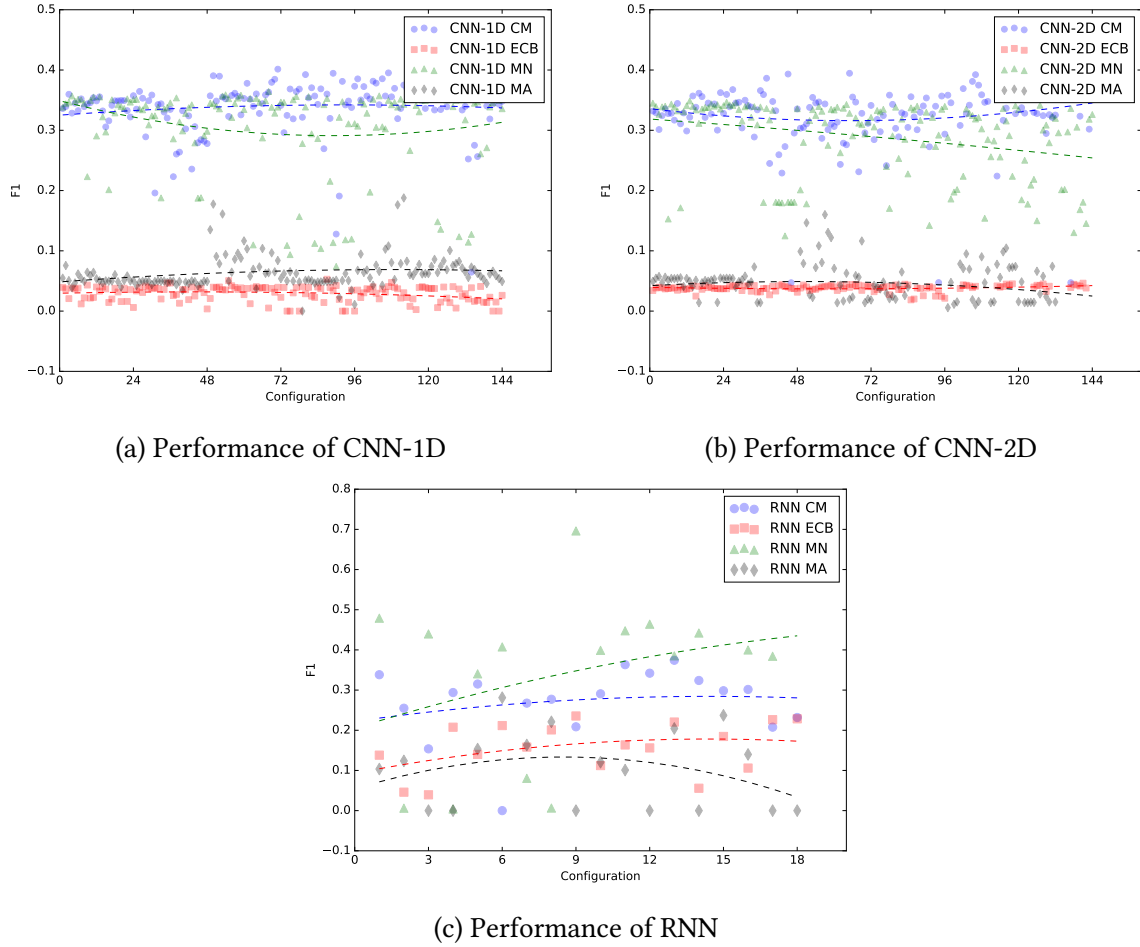


Figure 5.9: Scatter plots of the performance (F1) exhibit by the considered deep learning models along with their corresponding trendline

variance among the F1 scores obtained at different configurations. Though, CNN-1D shows lower variance, the model has higher number of outliers compared to CNN-2D model. RNN model performs significantly superior compared to convolution models for *empty catch block* smell with an F1 score of 0.22 versus 0.04 and 0.02 achieved by CNN-1D and CNN-2D respectively; the performance of the model, however, shows a wide variation depending on the chosen hyper-parameters. For *magic number* smell, most of the RNN configurations do better than the best of the convolution-based configurations. RNN exhibits a very high variance in the performance compared to convolution models for *multifaceted abstraction* smell.

Equipped with experiment results, we attempt to validate the hypotheses. We present AUC, precision, recall, and F1 to show the performance of the analyzed deep learning models. We attempt to validate each of the addressed hypotheses in the rest of the section.



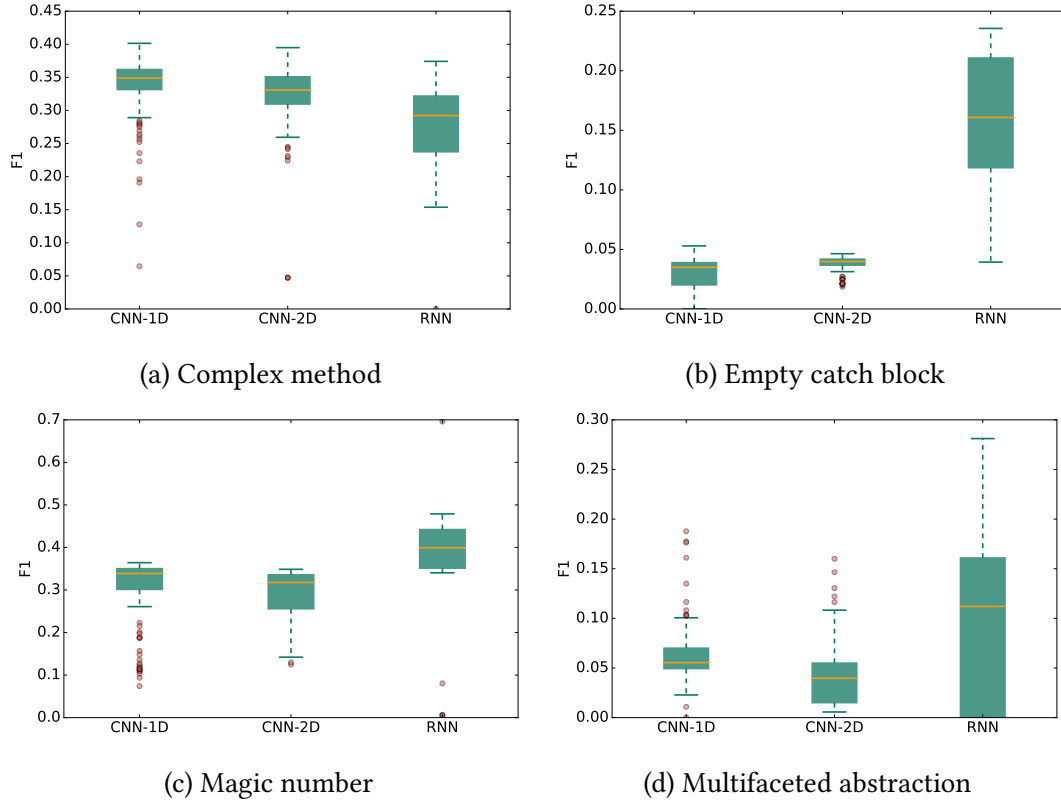


Figure 5.10: Boxplots of the performance (F1) exhibit by the considered deep learning models for all the four smells

5.2.1.1 D-RQ1.H1. It is feasible to detect smells using deep learning methods.

Table 5.7 lists performance metrics (AUC, precision, recall, and F1) for the optimal configuration for each smell, comparing all three deep learning models. It also lists the hyper-parameters associated with the optimal configuration for each smell. Figure 5.11 presents the performance (F1) of the deep learning models corresponding to each smell considered in this exploration.

For *complex method* smell, CNN-2D performs the best; though, performance of CNN-1D is comparable. This could be an implication of the fact that the smell is exhibited through the structure of a method; hence, CNN models, in this case, could identify the related structural features for classifying the smells correctly. On the other hand, CNN models perform significantly poorer than RNN in identifying *empty catch block* smells. The smell is characterized by a micro-structure where catch block of a try-catch statement is empty. RNN model identifies the sequence of tokens (*i.e.*, opening and closing braces), following the tokens of a try block, whereas CNN models fail to achieve that and thus RNN performs significantly better than the CNN models. Also, the RNN model performs remarkably better than CNN models for *magic number* smell. The smell is characterized by a specific range of tokens and the RNN does well in spotting them. *Multifaceted abstraction* is a non-trivial smell that requires analysis of method interactions to observe incohesiveness of a class. None of the employed deep learning models could capture the complex characteristics of the smell, implying that the

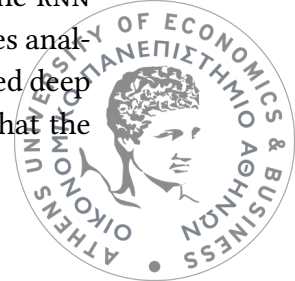


Table 5.7: Performance of all three models with configuration corresponding to the optimal performance. L refers to deep learning layers, F refers to number of filters, K refers to kernel size, MPW refers to maximum pooling window size, ED refers to embedding dimension, LSTM refers to number of LSTM units, and E refers to number of epochs.

	Performance					Configuration						
	Smells	AUC	Precision	Recall	F1	L	F	K	MPW	ED	LSTM	E
CNN-1D	CM	0.82	0.26	0.69	0.38	2	16	7	4	–	–	25
	ECB	0.59	0.02	0.31	0.04	2	64	11	4	–	–	40
	MN	0.68	0.18	0.77	0.29	2	16	5	5	–	–	17
	MA	0.83	0.05	0.75	0.09	3	16	11	5	–	–	36
CNN-2D	CM	0.82	0.30	0.68	0.41	3	64	5	4	–	–	17
	ECB	0.50	0.01	1	0.02	3	64	7	2	–	–	32
	MN	0.65	0.31	0.41	0.35	1	16	11	2	–	–	50
	MA	0.87	0.03	0.95	0.06	2	8	7	2	–	–	19
RNN	CM	0.85	0.19	0.80	0.31	3	–	–	–	16	32	8
	ECB	0.86	0.13	0.76	0.22	2	–	–	–	16	128	15
	MN	0.91	0.55	0.91	0.68	2	–	–	–	16	128	19
	MA	0.69	0.01	0.86	0.02	1	–	–	–	32	128	9

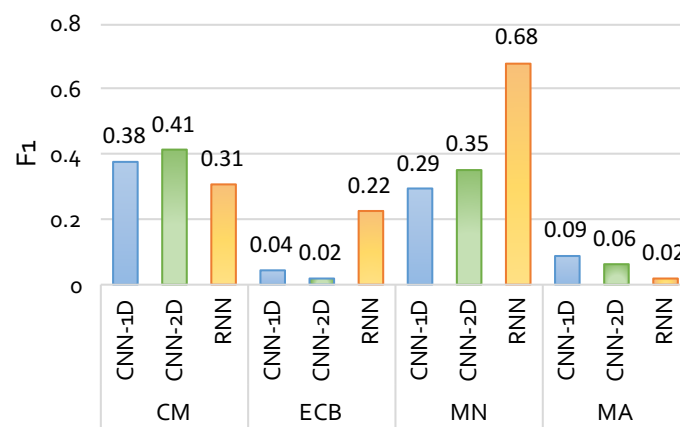
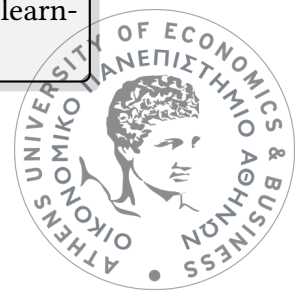


Figure 5.11: Comparative performance of the deep learning models for each considered smell

token-level representation of the data may not be appropriate for capturing higher-level features required for detecting the smell. It is evident from the above discussion that all the employed models are capable of detecting smells in general; however, their smell-specific performances differ significantly.

Therefore, the hypothesis exploring the feasibility of detecting smells using deep learning models holds true.



5.2.1.2 D-RQ1.H2. CNN-2D performs better than CNN-1D in the context of detecting smells.

Table 5.7 shows that CNN-1D performs better than CNN-2D model for *empty catch block* and *multifaceted abstraction* smells with optimal configuration. On the other hand, CNN-2D performs slightly better than its one dimension counterpart for detecting *complex method* and *magic number* smells. In summary, there is no universal superior model for detecting all four smells; their performance varies depending on the smell under analysis.



Therefore, we reject the hypothesis that CNN-2D performs overall better than CNN-1D as none of the models is clearly superior to another in all the cases.

5.2.1.3 D-RQ1.H3. RNN model performs better than CNN models in the smell detection context.

Table 5.8 presents the comparison of RNN with CNN-1D and CNN-2D by comparing pairwise F1 measure differences in percentages, where the F1 values are obtained by the optimal configuration in each case. Here, the performance difference in percentage is calculated by $(F1_{RNN} - F1_{CNN})/F1_{RNN} \times 100$. RNN performs far superior for *empty catch block* and *magic number* smells against both convolution models. However, the performance of RNN is lower for *complex method* and *multifaceted abstraction* smells.

Table 5.8: Performance (F1) comparison of RNN with CNN-1D and CNN-2D

Smell	RNN vs CNN-1D	RNN vs CNN-2D
CM	-22.94%	-33.81%
ECB	80.23%	91.94%
MN	57.19%	48.48%
MA	-353.15%	-208.00%



The analysis suggests that performance of the deep learning models is smell-specific. Therefore, we reject the hypothesis that RNN models perform better than CNN models for all considered smells.



Implications

This is the first attempt in the software engineering literature to show the feasibility of detecting smells using deep learning models from the tokenized source code without extensive feature engineering. It may motivate researchers and developers to explore this direction and build over it. For instance, *context* plays an important role in deciding whether a reported smell is actually a quality issue for the development team. One of the future works that the community may explore is to combine the models



trained using samples classified by the existing smell detection tools with the developer's feedback to identify more relevant smells considering the context.

Our results show that, though both convolution methods perform superior for specific smells, their performance is comparable for each smell. This implies that we may use one-dimensional or two-dimensional CNN interchangeably without compromising the performance significantly.

The comparative results on applying diverse deep learning models for detecting different types of smells suggest that there exists no universal optimal model for detecting all smells under consideration. The performance of the model is highly dependent on the kind of smell that the model is trying to classify. This observation provides grounds for further investigation, encouraging the software engineering community to propose improvements on smell-specific deep learning models.

5.2.2 D-RQ2. Is transfer-learning feasible in the context of detecting smells? If yes, which deep learning model exhibits superior performance in detecting smells when applied in transfer-learning setting?

We explore the feasibility of applying transfer-learning in smell detection context. If it is feasible, we are interested to learn which deep learning model exhibits superior performance.

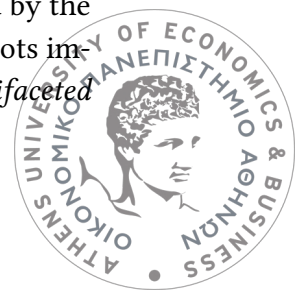
Approach

In the case of direct-learning, the training and evaluation samples belong to the same programming language whereas in the transfer-learning case, the training and evaluation samples come from two similar but different programming languages. This research question inquires the feasibility of applying transfer-learning *i.e.*, train neural networks by using C# samples and employ the trained model to classify code fragments written in Java.

For the transfer learning experiment we keep the training samples exactly the same as the ones we used in RQ1. For evaluation, we download repositories containing Java source code and preprocess the samples as described in Section 4.2.1. Similar to RQ1, evaluation is performed on a realistic scenario, *i.e.*, we use all the positive and negative samples from the selected repositories. This arrangement ensures that the models would perform as reported if employed in a real-world application. Table 5.9 shows the number of samples used for training and evaluation for this research question.

Results

As an overview, Figure 5.12 shows the scatter plots for each deep learning model comparing the performance (F1) of both the direct-learning and transfer-learning for all the considered smells for all the configurations. These plots outline the performance exhibited by the models in both the cases with trend lines distinguishing the compared series. The plots imply that the models perform better in the transfer-learning case for all except *multifaceted abstraction* design smell.



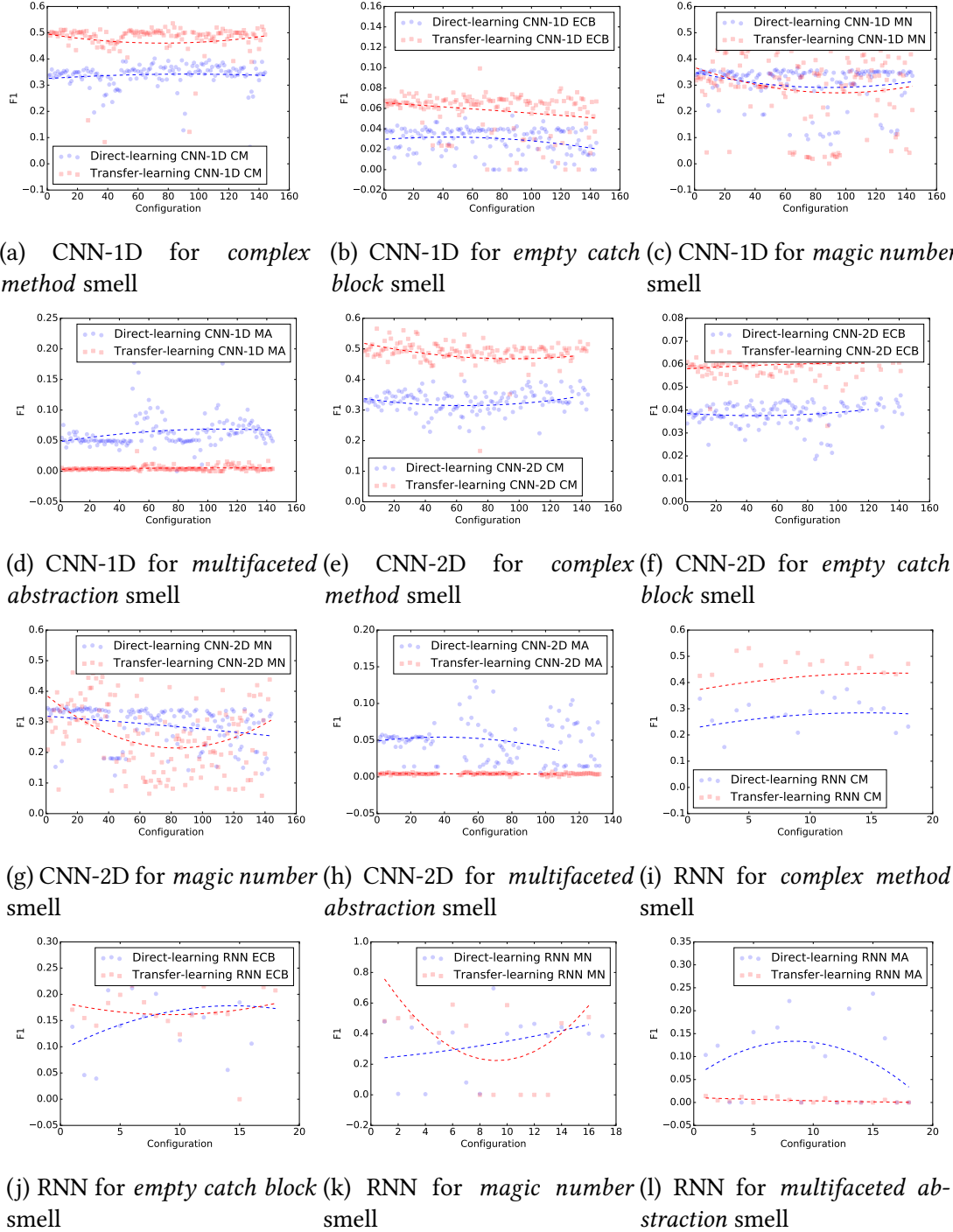


Figure 5.12: Scatter plots for each model and for each considered smell comparing F1 of direct-learning and transfer-learning along with corresponding trendline



Table 5.9: Positive (P) and negative (N) number of samples used for training and evaluation for RQ2

	1-D			2-D		
	Training P and N	Evaluation P	N	Training P and N	Evaluation P	N
CM	3,472	2,163	48,633	2,641	2001	30,215
ECB	1,200	597	50,199	982	538	31,678
MN	5,000	42,037	50,905	5,000	7,778	24,438
MA	290	25	13,110	284	23	11,812

In the rest of the section, we report quantitative results on applying transfer learning between C# to Java. The results are based on the optimal configuration of each model for each smell.

5.2.2.1 D-RQ2.H1. It is feasible to apply transfer-learning in the context of code smells detection.

Table 5.10 presents the performance of the models for all the considered smells demonstrating strong evidence on the feasibility of applying transfer-learning for smell detection. The performance pattern is in alignment to that in the direct-learning case; Spearman correlation between the performance produced by direct-learning and transfer-learning is 0.98 (with p-value = 1.309×10^{-8}).

Therefore, we accept the hypothesis that transfer-learning is feasible in the context of code smells detection.

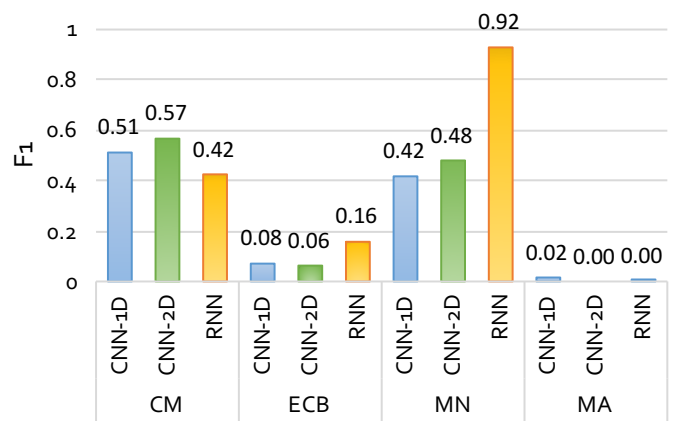


Figure 5.13: Comparative performance of the deep learning models for each considered smell in transfer-learning settings

Figure 5.13 presents a comparison among the performance (i.e., F1) exhibited by all the deep learning models for each considered smell. RNN performs significantly superior for



Table 5.10: Performance of all three models with configuration corresponding to the optimal performance. L refers to deep learning layers, F refers to number of filters, K refers to kernel size, MPW refers to maximum pooling window size, ED refers to embedding dimension, LSTM refers to number of LSTM units, and E refers to number of epochs.

	Performance					Configuration						
	Smells	AUC	Precision	Recall	F1	L	F	K	MPW	ED	LSTM	E
CNN-1D	CM	0.87	0.38	0.79	0.51	2	32	7	4	–	–	23
	ECB	0.56	0.05	0.15	0.08	3	8	5	5	–	–	27
	MN	0.64	0.48	0.37	0.42	1	32	11	3	–	–	12
	MA	0.52	0.01	0.04	0.02	2	8	11	5	–	–	13
CNN-2D	CM	0.88	0.43	0.84	0.57	1	8	7	2	–	–	37
	ECB	0.54	0.04	0.12	0.06	3	16	5	4	–	–	19
	MN	0.65	0.43	0.54	0.48	1	64	5	4	–	–	8
	MA	0.50	0.0	0.0	0.0	3	8	5	5	–	–	17
RNN	CM	0.66	0.62	0.32	0.42	1	–	–	–	32	64	8
	ECB	0.90	0.09	0.91	0.16	3	–	–	–	32	32	27
	MN	0.95	0.94	0.91	0.92	1	–	–	–	32	32	22
	MA	0.51	0.0	0.08	0.0	1	–	–	–	32	32	18

empty catch block and *magic number* smells following a trend comparable to direct-training. For *complex method* smell, CNN-2D performs the best followed by CNN-1D. All the three models perform poorly with *multifaceted abstraction* smell.

5.2.2.2 D-RQ2.H2. Transfer-learning performs inferior compared to direct learning.

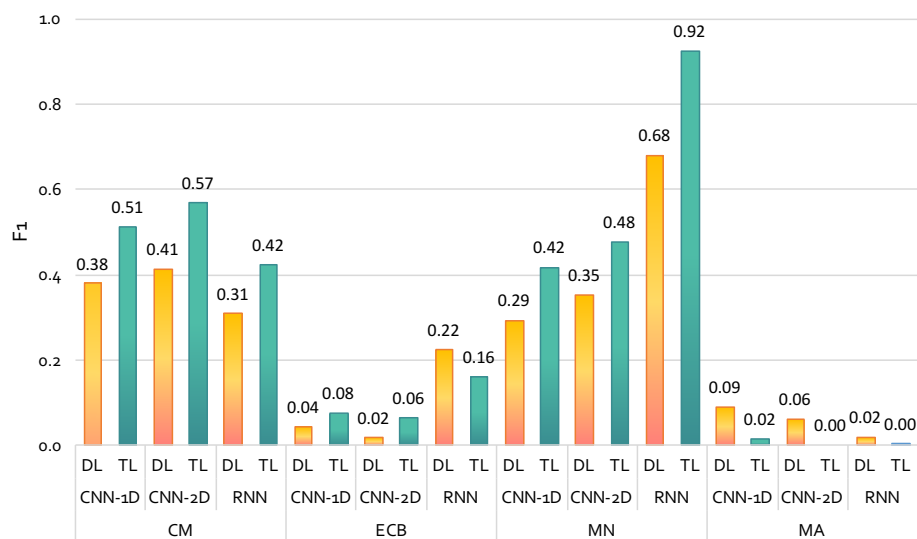
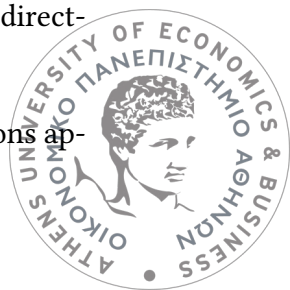


Figure 5.14: Comparison of performance of the deep learning models between direct-learning (DL) and transfer-learning (TL) settings

Figure 5.14 compares the performance of the models at their optimal configurations ap-



plied in the transfer-learning and in direct-learning. We observe that, in majority of cases, transfer-learning performs better than the corresponding direct-learning counterpart models. The only exception for implementation smells is RNN applied on *empty catch block* smell, where direct-learning shows better results. For the only design smell, i.e., *multifaceted abstraction*, all the models perform poorly in both cases.

To dig deeper into the cause of better performance of deep learning models in the transfer-learning case, we calculate the ratio of positive and negative evaluation samples in both research questions. Table 5.11 presents the ratio for samples used in both the research questions as well as percentage difference of the ratios of positive and negative samples in RQ2 compared to the sample ratio in RQ1. The percentage difference is computed as follows: $(Ratio_{RQ2} - Ratio_{RQ1}) / Ratio_{RQ1} \times 100$. It is evident that Java code samples have higher ratio of positive samples, up to 188% higher, compared to C# samples for implementation smells. We deduce that due to significantly higher number of positive samples, the deep learning models show better performance statistics in the transfer-learning case. On the other hand, *multifaceted abstraction* smell occurs significantly lower (up to 72%) in Java code compared to C# code and this lower ratio further degrades the performance of the models for *multifaceted abstraction* smell.



Therefore, due to size discrepancies in the samples available for evaluation in direct-learning and transfer-learning, we cannot conclude the superiority or inferiority of the results produced by applying transfer-learning compared to those of direct-learning.

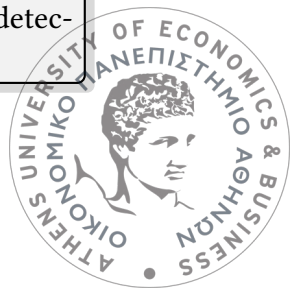
Table 5.11: Difference in ratio (in percent) of positive and negative evaluation samples in RQ2 compared to sample ratio in RQ1

Smell	Ratio (RQ1)		Ratio (RQ2)		Difference %	
	1D	2D	1D	2D	1D	2D
CM	0.0287	0.0250	0.0445	0.0662	35.53	62.19
ECB	0.0097	0.0092	0.0119	0.0170	18.14	45.88
MN	0.1242	0.1210	0.2084	0.3183	40.40	61.98
MA	0.0055	0.0070	0.0019	0.0019	-188.42	-260.87



Implications

Our results demonstrate that it is feasible to apply transfer-learning in the smell detection context. Exploiting this approach can lead to a new category of smell detection tools, specifically for the programming languages where comprehensive smell detection tools are not available.



5.2.3 Discussion

As is the case with most research, our results are sobering rather than sensational. Although it is possible to detect some code smells using deep learning models, the method is by no means universal, and the outcome is sensitive to the training set composition and the training time. In the rest of the section, we elaborate on these observations emerging from the presented results.

5.2.3.1 Is there any silver-bullet?

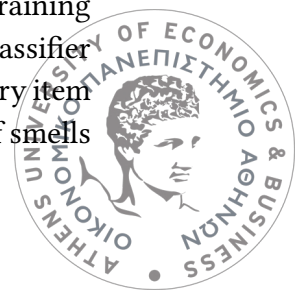
In practical settings one would want to employ a universal deep learning model that performs well for a variety of smells. In addition, a universal model architecture that performs consistently well for all the considered smells would allow the implementation of tools simpler.

RNN has the reputation to perform well with textual data and sequential patterns while CNN is considered good for imaging data and visual patterns. Given the similarity of source code and natural language, it is expected to obtain good performance from RNN. Our results show that RNN significantly outperforms both CNN models in the cases of *empty catch block* and *magic number*. However, in the case of *complex method*, the CNN models outperform the RNN whereas in the case of *multifaceted abstraction*, none of the models yield satisfactory results. These outcomes suggest that there is not one deep learning model that can be used for all kinds of smells. We have a uniform model architecture for each model and we observed that the performance of the model differs significantly for different smells. It suggests that it is non-trivial, if not impossible, to propose a universal model architecture that works for all smells. Each smell exhibits diverse distinctive features and hence their detection mechanisms differ significantly. Therefore, given the nature of the problem, it is unlikely that one universal model architecture will be the silver-bullet for the detection of a wide range of smells.

5.2.3.2 Performance comparison with baseline

It is not feasible to compare the results presented in this paper with other attempts [KVGS09, KVGS11, MAB⁺12b, MAB⁺12a, BBEAM10, BKG19, FPRZ16] that use machine learning for smell detection due to the following reasons. First, the replication packages of the related attempts are not available. Second, for most of the existing attempts, the ratio of positive and negative evaluation samples is not known; in the absence of this information, we cannot compare them with our results fairly since the ratio plays an important role in the performance of machine learning models. Furthermore, the existing approaches compute metrics and feed them to machine learning models while we feed tokenized source code.

We compare our results with the results obtained from two baseline random classifiers that do not really learn from the data but use only the distribution of smells in the training set to form their predictions. Table 5.12 presents the comparison. The first random classifier generates predictions by following the training set's class distribution: that is, for every item in the evaluation set it predicts whether it is a smell or not based on the frequency of smells



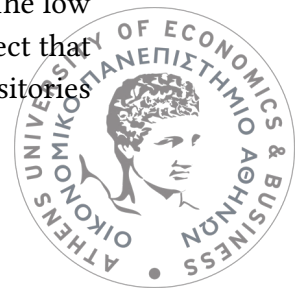
in the training data. We did that for both balanced and unbalanced evaluation samples to mimic the learning process of the actual experiment. In the middle three columns, referred to as ‘RC (*frequency*)’, of the table we show the results for the balanced setting, as they were better than the results for the unbalanced setting. The second random classifier predicts always that a smell is present; this gives perfect recall, but low precision, as you can see in the columns corresponding to ‘RC (*all smells*)’ of the table. Overall, our models perform far better than a random classifier for all but *multifaceted abstraction* smell for both baseline variants.

Table 5.12: Comparison of performance (Precision, Recall, and F1) with a random classifier (RC) following the training set frequencies or responding always indicating a smell

	Smells	Our results			Performance RC (<i>frequency</i>)			RC (<i>all smells</i>)		
		P	R	F1	P	R	F1	P	R	F1
CNN-1D	CM	0.38	0.79	0.51	0.03	0.50	0.05	0.03	1	0.05
	ECB	0.05	0.15	0.08	0.01	0.50	0.02	0.01	1	0.02
	MN	0.48	0.37	0.42	0.11	0.50	0.18	0.11	1	0.20
	MA	0.01	0.04	0.02	0.01	0.50	0.01	0.01	1	0.01
CNN-2D	CM	0.43	0.84	0.57	0.02	0.50	0.05	0.02	1	0.05
	ECB	0.04	0.12	0.06	0.01	0.50	0.02	0.01	1	0.02
	MN	0.43	0.54	0.48	0.11	0.50	0.18	0.11	1	0.19
	MA	0.0	0.0	0.0	0.01	0.50	0.01	0.01	1	0.01
RNN	CM	0.62	0.32	0.42	0.03	0.50	0.05	0.03	1	0.05
	ECB	0.09	0.91	0.16	0.01	0.50	0.02	0.01	1	0.02
	MN	0.94	0.91	0.92	0.11	0.50	0.18	0.11	1	0.20
	MA	0.0	0.08	0.0	0.01	0.50	0.01	0.01	1	0.01

5.2.3.3 Poor performance in detecting a design smell

The presented neural networks perform very poor when it comes to detecting the sole design smell *multifaceted abstraction*. We infer the following two reasons for this under-performance. First, design smells such as *multifaceted abstraction* are inherently difficult to spot unless a deeper semantic analysis is performed. Specifically, in the case of *multifaceted abstraction*, interactions among methods of a class as well as the member fields are required to observe cohesion among the methods which is a non-trivial aspect and the neural networks could not spot this aspect with the provided input. Therefore, we need to provide refined semantics information in the form of engineered features along with the source code to help neural networks identify the inherent patterns. Second, the number of positive training samples were very low, thus significantly restricting our training set. The low number severely impacts the ability of neural networks to infer the responsible aspect that cause the smell. This limitation can be addressed by increasing the number of repositories under analysis.



5.2.3.4 Trading performance with training-time

As observed in the results section, RNN performs significantly superior than CNN in some specific cases. However, we also note that RNN models take considerable more time to train compared to CNN models. We log the time taken by each experiment for the comparison. Table 5.13 presents the average time taken by each model for each smell per epoch. The table shows that the RNN performance is coming from much more intense processing compared to CNN. Therefore, if the performance of RNN and CNN is comparable for a given task, one should go with CNN-based solution for significantly faster training time.

Table 5.13: Average training-time taken by the models to train a single epoch in seconds

	CNN-1D	CNN-2D	RNN
CM	1.2	1.0	2,134
ECB	0.8	0.5	1,038
MN	3.2	3.9	5,737
MA	0.8	4.6	2,208



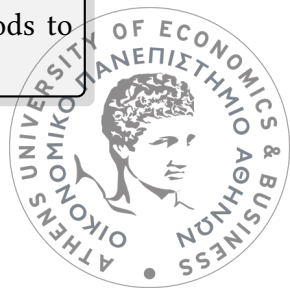
Opportunities

The study may encourage the research community to explore the deep learning methods as a viable option for addressing the problem of smell detection. Given that we did not consider the context and developers' opinion on smells reported by deterministic tools, it would be acutely interesting to combine these aspects either by considering the developers' opinion (by manually tagging the samples) while segregating positive and negative samples or by designing the models that take such opinions as an input to the model.

We have shown that transfer-learning is feasible in the context of code smells. This result introduces new directions for automating smell detection which is particularly useful for programming languages for which smell detection tools are either not available or not matured.

Though this work shows the feasibility of detecting implementation smells; however, complex smells such as *multifaceted abstraction* require further exploration and present many open research challenges. The research community may build on the results presented in this study and further explore optimizations to the presented models, alternative models, or innovative model architectures to address the detection of complex design and architecture smells.

Beyond smell detection, proposing an appropriate refactoring to remove a smell is a non-trivial challenge. There have been some attempts [TME⁺18, BSH⁺11] to separate refactoring changes from bug fixes and feature additions. One may exploit the information produced from such tools and the power of deep learning methods to construct tools that propose suitable refactoring mechanism.



5.3 Results of Maintainability Analysis on Configuration Code

This section presents the results gathered from the analysis of configuration code and our observations *w.r.t.* each research question addressed.

We use the term “total detected smells (by volume)” to refer to all the smell instances detected in a project. We use the term “total detected smells (by existence)” to refer to the number of different types of smell instances detected in a project. Additionally, we refer to each cataloged configuration smell as a three letter acronym as defined in the Section 3.2.3.

5.3.1 C-RQ1. What is the distribution of maintainability smells in configuration code?

Approach

We compute the total number of detected smell instances (by volume and by existence) for all the smells belonging to both implementation and design configuration smells categories.

Results

The left pane of Table 5.14 shows the number of smell instances detected for implementation configuration smells (ICS) both by volume ($I(V)$) and by existence ($I(E)$). The three most frequently occurring smells by volume and by existence are IIA (*improper alignment*), IQU (*improper quote usage*), and ILS (*long statement*). Similarly, IDE (*duplicate entry*), IMD (*missing default case*), INC (*inconsistent naming convention*) are some of the least frequently occurring smells.

The right pane of Table 5.14 shows the similar distribution for detected design configuration smells (DCS). The most frequently occurring design configuration smells are DIM (*insufficient modularization*) and DMF (*multifaceted abstraction*). Similarly, the least frequently occurring smells are DBH (*broken hierarchy*) and DDE (*deficient encapsulation*).

A few observations from the above table are the following.

- There is a relatively large number of smell instances reported for DDB (*duplicate block*) by volume; however, the smell only occurs in less than one forth of the analyzed repositories. **This indicates that either the developers of Puppet repositories do not duplicate the code at all or they do it massively.**
- Although, investigating and establishing the potential reasons of identified smell instances is not in the scope of this study, the nascent maturity phase of current configuration systems could be a cause for a few smells. Specifically, the support for system configuration code in terms of better tools and IDEs is still maturing which could potentially help avoiding smells such as IIA (*improper alignment*).

The reported frequently occurring smells may also motivate efforts in the future to identify their causes and steps to avoid them. Such efforts may focus on improving existing documentation, enhancing language support, and developing new tools.



Table 5.14: Distribution of Detected Implementation and Design Configuration Smells

Impl. config. smells	#I(V)	#I(E)	Design config. smells	#I(V)	#I(E)
Missing default case	4,604	706	Multifaceted abstraction	64,266	4,339
Inconsistent naming convention	4,804	440	Unnecessary abstraction	4,319	1,427
Complex expression	3,994	963	Imperative abstraction	4,354	1,575
Duplicate entity	65	29	Missing abstraction	1,913	813
Misplaced attribute	22,976	1,383	Insufficient modularization	96,033	4,422
Improper alignment	780,265	3,064	Unstructured module	4,653	3,337
Invalid property value	14,360	729	Duplicate block	17,601	1,016
Incomplete tasks	11,071	1,467	Broken hierarchy	83	37
Deprecated statement usage	6,466	674	Dense structure	1760	1760
Improper quote usage	428,951	2,463	Deficient encapsulation	1,075	424
Long statement	527,637	4,115	Weakened modularity	13,944	2,890
Incomplete conditional	4,797	1,217			
Unguarded variable	71339	1,405			

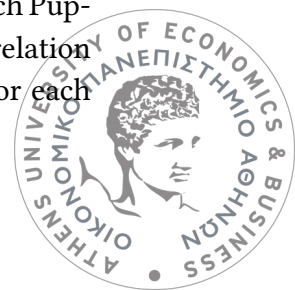
- It is interesting to note that DDS (*dense structure*) falls in the least occurring smell category by volume but most frequently occurring smell category by existence. It is due to the fact that there could be only one instance at the most for this smell in a project.

DUM (*unstructured module*) also exhibits similar characteristics. Since the tool analyzes the structure of a module as a whole, DUM gets identified at the most once for a module. Since each project deals with only a limited number of modules, the detected smell instances are relatively low whereas the smell occurred in relatively large number of analyzed projects.

5.3.2 C-RQ2. What is the relationship between the occurrence of design configuration smells and implementation configuration smells?

Approach

We count the total number of implementation and design configuration smells for each Puppet repository both by volume and by existence. Next, we compute Spearman correlation coefficient between the counted implementation and design configuration smells for each repository by volume and by existence.



Results

Figure 5.15a presents a scatter graph (with alpha set to 0.3) showing the co-occurrence between implementation and design configuration smells by volume. The figure shows a dense accumulation towards the left-bottom indicating that implementation and design configuration smells co-occur for relatively small number of identified smell instances.

Figure 5.15b shows a density graph showing the co-occurrence between implementation and design configuration smells by existence. The figure reveals a dense correlation between implementation and design configuration smells (by existence) in the left bottom quadrant of the figure where the number of identified smell types is half or less.

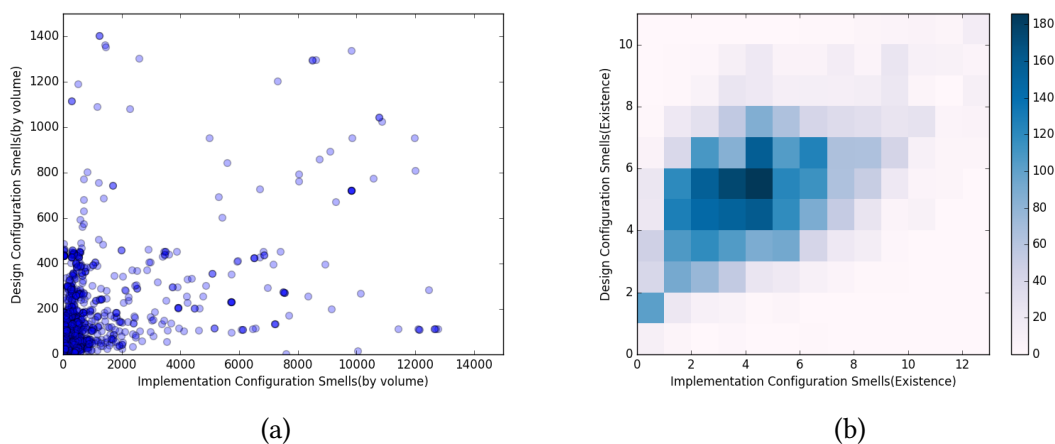


Figure 5.15: Co-occurrence between implementation and design configuration smells by (a) volume and by (b) existence

We compute Spearman correlation coefficient for both the cases. Table 5.15 shows the correlation coefficients and associated p-values. Both the analyses show positive correlation between implementation and design configuration smells with high statistical significance; however, correlation analysis by volume exhibits stronger correlation than correlation analysis by existence.

It shows that high volume of design (or implementation) configuration smells is a strong indication of the presence of high volume of implementation (or design) configuration smells in a project. Whereas, a project that shows presence of large number of design (or implementation) configuration smell types moderately indicates presence of large number of implementation (or design) configuration smell types.

5.3.3 C-RQ3. Is the principle of coexistence applicable to smells in configuration projects?

Approach

To compute intra-category co-occurrence for a smell, we count the number of occurrences



Table 5.15: Results of Correlation Analysis

	Correlation(ρ)	p-value
Analysis by volume	0.66410	$<2.2e-16$
Analysis by existence	0.44526	$<2.2e-16$

of other smells in the same category (by existence), only when the smell occurred. We evaluate the average co-occurrence for each smell across all the repositories considering only those values where the smell has occurred. We compute the average co-occurrence for all the implementation and design configuration smells and compared their normalized values.

Results

Figure 5.16 presents the average co-occurrence computed for each smell for both the implementation and design configuration smells. IDE (*duplicate entity*) with average 0.75 and IQU (*improper quote usage*) with average 0.29 are the implementation configuration smells that show the highest and lowest co-occurrences respectively in the category. In the design configuration smells category, DBH (*broken hierarchy*) with average 0.73 and DIM (*insufficient modularization*) as well as DMF (*multifaceted abstraction*) with average 0.36 show the highest and lowest co-occurrences respectively.

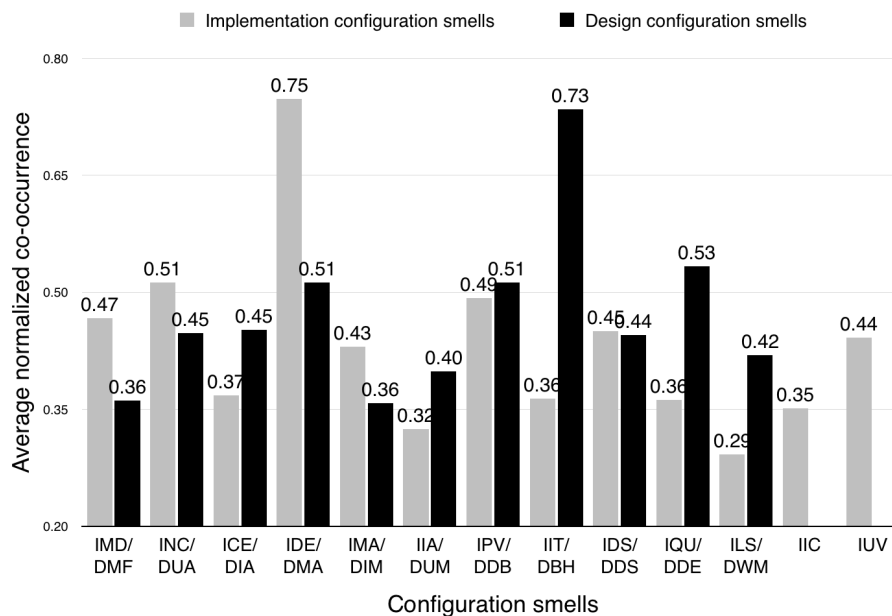


Figure 5.16: Average co-occurrence (intra-category) for implementation and design configuration smells



↗

The result implies that whenever *duplicate entity* or *broken hierarchy* smells are found, it is very likely to find other smells from the same category in the project. Whereas, the smells *improper quote usage* and *insufficient modularization* occur more independently.

Average normalized correlation for implementation and design configuration smells is 0.43 and 0.47 respectively. This leads to another interesting observation: **design configuration smells show 9.3% higher co-occurrence among themselves than the implementation configuration smells**. Since a design decision impacts the software in many ways, it is believed that one wrong or non-optimal design decision introduces many quality issues. The statistics reported above affirm the belief.

5.3.4 C-RQ4. Does smell density depend on the size of the configuration project?

Approach

We compute normalized smell density for both the smell categories for all the repositories and plot scatter graphs between lines of code in the repository and the smell density. We then perform correlation analysis on both sets and document our observations based on the received results.

Results

Figure 5.17 presents the distribution of normalized smell density for implementation and design configuration smells against lines of code (with $\alpha = 0.3$).

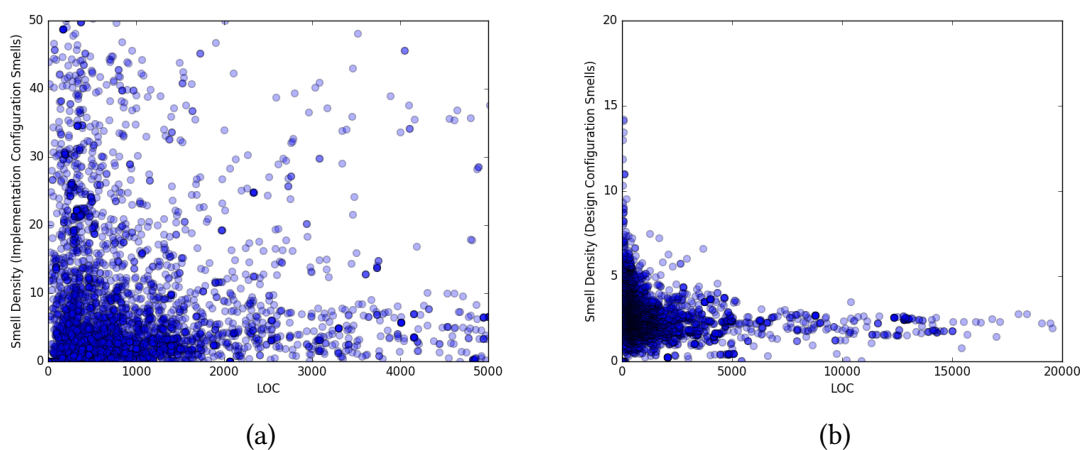


Figure 5.17: Smell density for (a) implementation configuration smells and (b) design configuration smells against lines of code

The visual inspection of the above graphs reveals the following observations.

- The distribution shown in Figure 5.17a is very scattered and random in comparison with the distribution in the Figure 5.17b. Although, both the figures show a weak



linear relationship between smell density and lines of code, Figure 5.17a shows a relatively stronger linear relationship than the Figure 5.17b.

- Large projects show maturity and tend to demonstrate lower smell density compared to smaller projects in both the cases.

We, then, compute Spearman correlation coefficient for both the data sets. The results of the correlation analysis are summarized in Table 5.16.

Table 5.16: Results of Correlation Analysis

	Correlation(ρ)	p-value
Implementation smells	0.03833	0.00980
Design smells	-0.32851	<2.2e-16

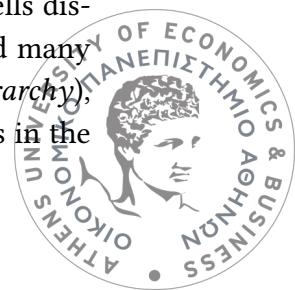


The results show no correlation between implementation smell density and size of the project. However, a weak negative correlation is perceived with high statistical significance between design smell density and the size of the project. This shows that as the project size increases, design configuration smell density tends to decrease. This result is interesting since it is believed traditionally that the complexity (and therefore, smell density) of a piece of software increases as the size of the software grows.

5.3.5 Discussion

Our empirical study reveals a large number of class declarations where the corresponding definitions are not found in the same repository. We find that 59% of the repositories that we analyze have at least one such instance. The majority of such missing definitions relate to third-party modules. A possible explanation is that software development teams exclude third-party modules from their Puppet code under version control. This practice provides the advantage of not having to maintain the used third-party modules as they change. However, it breaks the fundamental principle of IaC, i.e. production and configuration code should co-evolve. Such instances hurt the configuration process automation and are bound to lead to trouble in the form of missing dependencies. More interestingly, the Puppet language does not offer any solution to this problem since module installation, as opposed to package installation, cannot be part of a configuration code specification.

Yet another observation concerns language design (in this context Puppet). Diligent use of language features and adherence to best practices can drastically reduce smells discussed in this work. However, careful language design can also significantly avoid many configuration smells. For example, DUA (*unnecessary abstraction*), DBH (*broken hierarchy*), and DDE (*deficient encapsulation*) can be controlled and avoided by suitable changes in the



Puppet language. Similarly, many implementation configuration smells such as *IUV* (*un-guarded variable*), *IMA* (*misplaced attribute*), and *IDE* (*duplicate entity*) can also be checked at Puppet language level itself and can be avoided without any compromise in functionality and convenience.

5.4 Results of Maintainability Analysis on Database Code

In this section, we first present the results of a survey that we carried out to understand developers' perspective. We then present the results specific to each addressed research question. We also provide a discussion covering qualitative analysis and opportunities we perceive from this work.

5.4.1 Developers' Survey on Database Smells

We carried out an online survey targeting software developers to understand their perspective about the significance of various database schema smells. We divided the survey in three sections. In the first section, we collected information about participants' experience. In the second section, we asked the participants to read the description of each potential smell presented (total 13 questions based on the catalog presented in Section 3.2.4) and to rate each of them based on their *importance* (*i.e.*, the degree of smell's association with software quality issues), and *usefulness* (*i.e.*, the degree of accuracy of the smell in predicting software quality issues). All the questions in this section were Likert scale questions. We asked the respondents whether they consider the presented practice as a database schema smell, a recommended practice, both a smell and a recommended practice depending on the context, or neither a smell nor a recommended practice. The third section presented a couple of open-ended questions to get participants' view on the presented catalog and missing database schema smells. The questionnaire that we used is available online [Sha18a].

We ran a pilot for the survey, collected the feedback, and improved the survey. We shared the survey to all online social media channels and sought participation from the developer community. We received 52 complete responses out of 136 total responses.

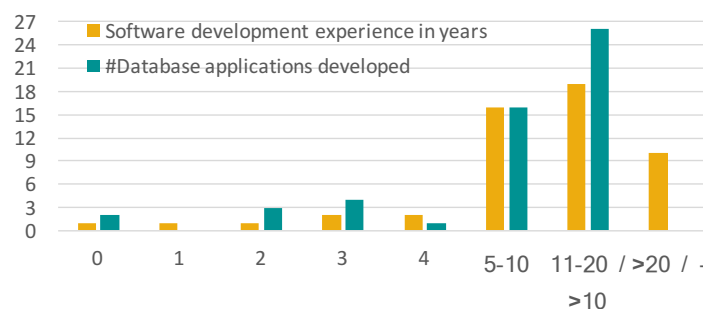


Figure 5.18: Experience of respondents in terms of number of years as well as the number of database applications developed by them



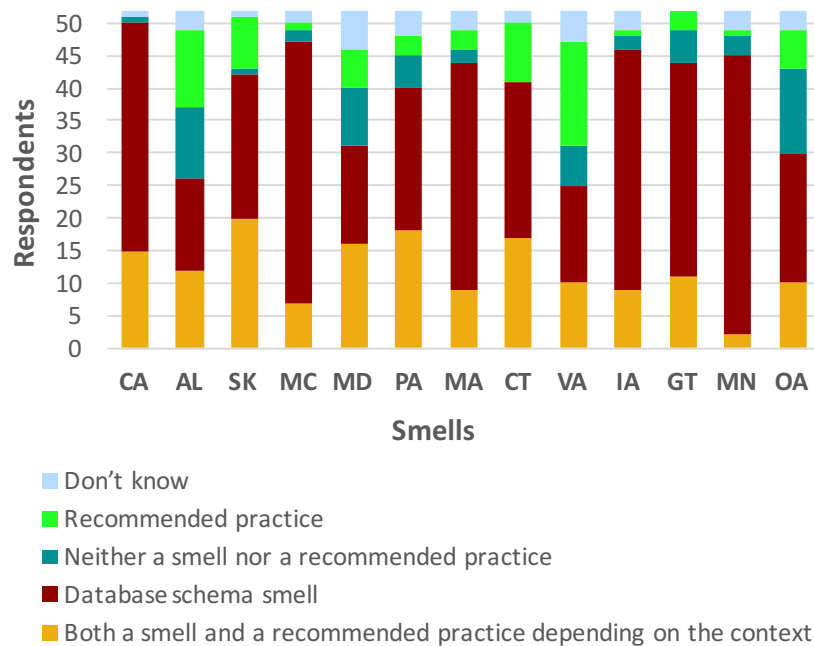
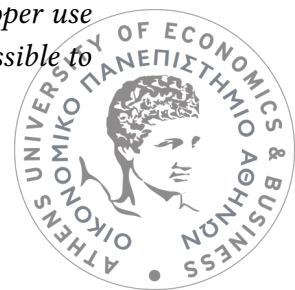


Figure 5.19: Respondents' perspective of considered database smells

Most of the respondents belong to experienced developer groups. Figure 5.18 shows the distribution of respondents' experience in terms of number of years and number of database applications they have developed. We summarize our findings from the survey below.

- A large majority of 88% agrees (42% strongly agree and 46% agree) that the awareness and knowledge of database smells is crucial for software developers to develop high quality applications. None of the respondents marked disagree or strongly disagree options.
- Figure 5.19 shows a consolidated perspective provided by the respondents for section 2 questions. Based on the responses we infer that some practices, such as *meaningless name* (83%) and *missing constraints* (77%), are clearly marked as database smells. However, we found that practices such as *values in attribute definition* and *adjacency list* are more context-sensitive.
- The respondents had the option to add their views either in terms of smells that we have not included but they have seen in practice as well as their feeling, objection, or reservation on the presented smells. A few respondents underline the subjectivity involved in database smell detection. For instance, one respondent said that “...database smells in general depend much more on an assessment of the need and end use of data...”. Similarly, another respondent shared an instance of duplicating values in a table (which is a smell) to avoid querying 60 tables to load a single record. Yet another respondent provided his/her opinion on *index abuse* smell: “...the proper use of indexes is dependent on many things and without regular profiling it's not possible to decide whether indexes are actually being misused.”



Summary of the survey

As a conclusion of our survey, developers seem to acknowledge the need for detecting database smells. However, their systematic identification remains an open problem. This points to the need for a tool that automatically detects the database smells. Developers may then, considering the context of the smell, decide whether the detected smells are indeed quality issues or serving a required purpose.

5.4.2 DB-RQ1. What are the occurrence patterns of database smells?

Approach

We use *DbDeo* to detect 9 types of database schema smells in the 357 industrial and 2568 open-source repositories. We collate all the detected instances of smells by their type and we compute average smell density for each type of smell.

Results

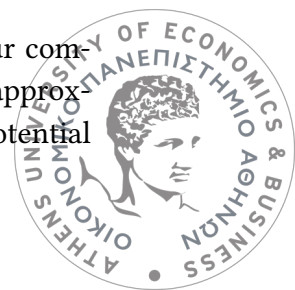
Table 5.17 summarizes the detected instances of database schema smells and corresponding average occurrences per repository in all the analyzed repositories.

Table 5.17: Occurrences of database schema smells for industry (I) as well as open-source (OSS) repositories

Smells	Occurrences		Avg. smell density	
	I	OSS	I	OSS
Compound attribute	5,517	7,966	0.04	0.04
Adjacency list	733	297	0.15	0.02
God table	4,428	5,507	0.44	0.24
Values in attribute definition	85	326	0.00	0.02
Metadata as data	944	1,003	0.16	0.09
Multi-column attribute	1,624	3,137	0.10	0.07
Clone table	101	3,704	0.00	0.05
Overloaded attribute names	1814	7,300	0.20	0.21
Index abuse	12,643	9,475	1.25	1.76

We make the following observations from the collected data in the context of this question. We find that *index abuse* is the most frequently occurring smell in both industrial as well as open-source projects. However, it is interesting to note that although the number of instances of *index abuse* smell are higher in industrial projects, they occur relatively less frequently than open-source projects considering their density. On the other hand, *values in attribute definition* in industrial projects and *adjacency list* in open-source projects are the least frequently occurring smells.

In industrial projects, some smells show significantly higher proneness to occur compared to open-source projects. For instance, smell density of *adjacency list* smell is approximately seven times higher in industrial projects than the open-source projects. A potential



reason of the observation is the higher size and complexity of the industrial projects. On the other hand, *clone table* tends to occur in open-source projects considerably more frequently than in industrial projects.

From the developers' survey, we learned that smells CA (*compound attribute*) and IA (*index abuse*) are the least subjective smells (*i.e.*, context matters the least for such smells) whereas smells AL (*adjacency list*) and VA (*values in attribute definition*) are most subjective in nature. This observation implies that a developer might be hesitant to introduce CA or IA and more open to adopt a solution that involves smells such as AL or VA. Interestingly, the occurrence patterns show exactly the opposite trend with respect to these smells; *i.e.*, smells CA and IA occur the most and smells AL and VA occur the least frequently in both industrial and open-source systems.

5.4.3 DB-RQ2. Does the size of the project or the database play a role in smell density?

Approach

We computed smell density for all the detected database smells. In this work, we define smell density as the number of database smells detected per 10 SQL statements. We then compute the Spearman correlation coefficient between total LOC (Lines Of Code) and smell density of the repository. We also compute the Spearman coefficient between size of the database (*i.e.*, number of CREATE TABLE statements) and smell density of the repository.

Results

The Spearman correlation coefficient (ρ) for the dataset is 0.2420 (p-value = 3.724×10^{-06}) for industrial projects and 0.0006 (p-value = 0.9731) for open-source projects. This indicates that density of database smells has low correlation for the industrial projects and no correlation for the open-source projects with the total lines of code in the repository.

We also explore the relationship between smell density and size of the database where size of a database is measured by the number of CREATE TABLE statements in the repository. The Spearman correlation analysis provides us $\rho = 0.7338$ (p-value < 2.2×10^{-16}) for industrial projects and $\rho = 0.6174$ (p-value < 2.2×10^{-16}) for open-source projects.



The values of the correlation coefficient show that smell density and size of the database share a fairly strong correlation *i.e.*, as the size of database increases, density of database smells tends to increase.



5.4.4 DB-RQ3. Does the nature of code (type of the application, or usage of ORM frameworks) affect the smell density?

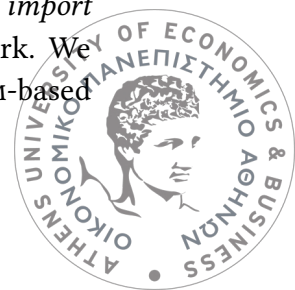
Approach

We extract information concerning the nature of subject systems; specifically, we infer the type of application and used ORM (Object-Relational Mapping) framework in each repository.

We infer the type of application among the following set — *Desktop*, *Mobile* (either *ios* or *Android*), or *Web*. We use the following heuristics to classify a repository to one of the application types.

- We figure out the programming language used primarily in a repository. To know the programming language used primarily in a repository, we scan all the files in the repository, detect the files containing source-code using their file extensions, and count the number of files for each programming language that we detect. We look for the following programming languages: ASP, C, C#, C++, HTML, Java, JavaScript, Objective c, PHP, Perl, Python, Ruby, SQL, VB, and XML.
- If the prime programming language is Java and there exists a manifest file with name ‘AndroidManifest.xml’, we conclude that the application is of type *Mobile(Android)*.
- If the prime programming language is Objective c, we tag the application as a *Mobile(ios)* application.
- If the repository contains one of the folders ‘Static’, ‘css’, or ‘public_html’ and primarily used programming language is one of the PHP, ASP, XML, or Python, then we classify the application type as *Web*.
- If the prime language is HTML, then also we interpret the application type as *Web*.
- If none of the above conditions are met for a repository, we classify it as a *Desktop* application.

Once we identify the type of all the repositories, we measure the average smell density for each application type. We select a list of 19 well-known ORM frameworks targeting different programming languages — C++ (LiteSQL, ODB, QxOrm), Java (ActiveJDBC, Apache Cayenne, Eclipse Link, Enterprise JavaBeans, Hibernate, Mybatis), Objective C (Core Data), C# (Dapper, Entity Framework, LINQ to SQL, NHibernate), PHP (Doctrine, Propel), and Python (SQLAlchemy, Django, SqlObject). We scan the dependencies of a repository specified in *import* (or similar) statements to detect whether the repository uses an ORM framework. For instance, we look at import statements in Java applications for the presence of *import org.apache.Cayenne* to infer that the application is using Apache Cayenne framework. We measure and compare the average smell density for both ORM-based and non-ORM-based repositories.



Results

Figure 5.20 (left) shows average smell density for different types of applications. The figure shows that 1998 open-source and 346 industrial repositories are classified as *Desktop*, 40 open-source and 2 industrial repositories as *Mobile*, and 530 open-source and 9 industrial repositories as *Web* applications. For open-source repositories, all three application types exhibit similar database schema smell density. This indicates that application type is not a significant factor that affects database smell density for open-source repositories. On the other hand, industrial *Web* applications show significantly lower smell density than the industrial *Desktop* applications although the sample for mobile and web applications in industrial projects is not significant from a statistical perspective.

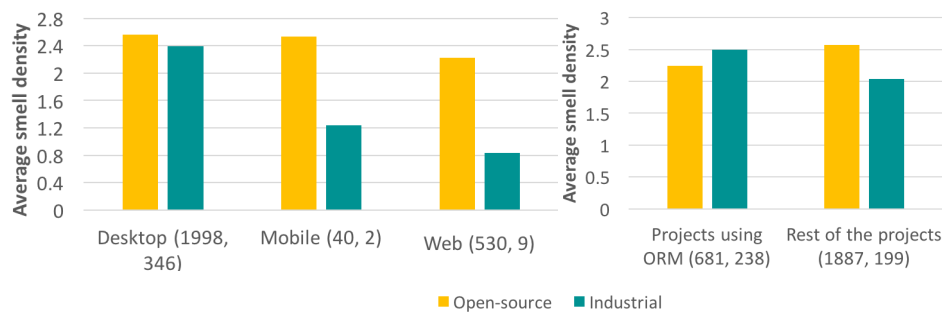


Figure 5.20: Average smell density of different types of applications (left) and projects using ORM frameworks and rest of the projects (right)

Right side of figure 5.20 shows average smell density for repositories separated based on whether they use an ORM framework or not. We observed that 681 open-source and 238 industrial projects use ORM frameworks among the analyzed projects. For industrial projects, non-ORM-based projects show lower average smell density than the projects based on ORM frameworks whereas we observe an opposite trend for open-source projects. However, Mann-Whitney U test shows that the difference in the average smell density is not statistically significant (p-value = 0.0252 for industrial and p-value = 0.1612 for open-source projects).



Thus, ORM frameworks do not bring immunity from database schema smells.

5.4.5 DB-RQ4. What is the degree of co-occurrence among database smells?

Approach

For each detected smell, we count occurrences of rest of the smells in the repository to investigate the degree of co-occurrence among database smells. We compute average co-occurrence for each smell across all the repositories. We take the average of the co-occurrences



taking into consideration only those values where the smell has occurred at least once. Further, we normalize the average co-occurrence values with number of detected smells. This exercise reveals the normalized co-occurrence patterns among database smells.

Results

Figure 5.21 shows average co-occurrence among database smells. The figure reveals that *clone table* for industrial projects and *values in attribute definition* for open-source projects show highest co-occurrence with other smells. *Index abuse* smell exhibits lowest co-occurrence with other smells for both the categories of projects. It implies that whenever a *clone table* in an industrial project or *values in attribute definition* smell in an open-source project occurs, it is very likely to find other database smells in the project. On the other hand, *index abuse* smell occurs more independently.

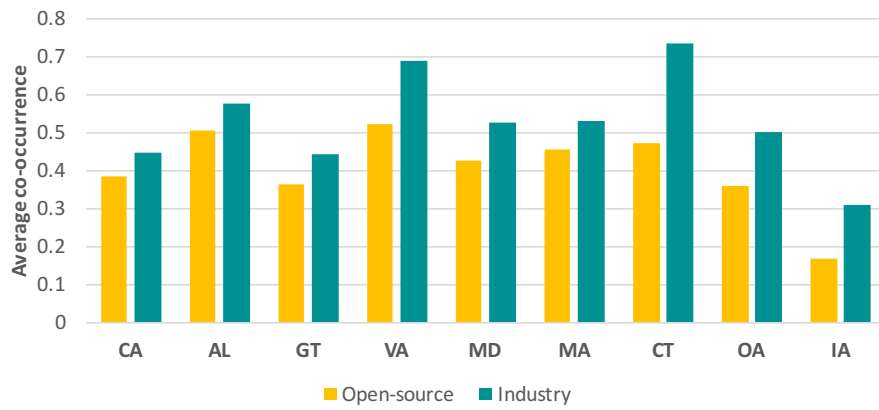


Figure 5.21: Average co-occurrence among database smells

Another interesting observation from figure 5.21 is that smells show considerably higher correlations in industrial projects. A potential reason of the fact could be the larger size of industrial projects than the open-source projects (industrial projects are five times larger on average in terms of LOC compared to open-source projects).

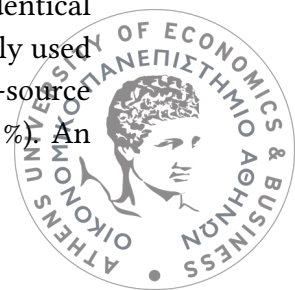
5.4.6 Discussion

In this section, we first discuss our observations about accuracy of the developed tool *DbDeo*. We also present our qualitative analysis of the results presented in Section 5.1.

5.4.6.1 Qualitative Analysis of the Results

In this section, we discuss the results obtained from our analysis presented in Section 5.1 from a qualitative perspective.

Our analysis found a considerable number of *overloaded attribute names* smells. Interestingly, many times developers declare attributes, even the primary keys, with identical names but with different types in a repository. We found that *ID* is the most popularly used name for a primary key. More than 40% of the analyzed tables belonging to open-source projects use *ID* as a primary key. For industrial projects, it is considerably lower (11%). An



interesting observation is that their type differs significantly. We found 13 and 12 different types being used for the attribute `ID` across all the analyzed open-source and industrial repositories respectively.

During manual exploration, we also observed one of the reasons for smells *clone table* and *overloaded attribute names* to occur. We observed that these smells occur often in test or example code. This observation highlights the quality deficit introduced in test or example code and possibly reveals the casual mindset of developers while writing test or example code.

Parameterized queries (where values or even sometimes attribute names are supplied dynamically) are very common for embedded SQL statements in source code. We observed `CREATE TABLE` statements are majorly defined statically; however, understandably, the majority of `SELECT` statements are defined as parameterized queries. This observation has an impact on *index abuse* smell. Our analysis reveals that more than 77% detected instances of *index abuse* smell belong to the third variant of the smell (i.e., unused indexes). When parameterized queries expect attribute names dynamically, our tool cannot identify the used attribute names and produce false-positive instances of *index abuse* smell.



Opportunities

In the context of this study, we outline possible ways to improve the state of scientific and industrial practice.

Tool support: IDEs can provide support, native or extended (via plug-ins), for SQL statements. This may allow developers to spot common problems, such as *index abuse* and *multicolumn attribute*, early on and rectify them. Along the same lines, ORM frameworks may raise an alarm, for instance in the form of warnings, to attract developers' attention towards potential flaws in the database design. Sophisticated external tools may extend their support to detect database smells and improve the quality of database schemas. Further, language extensions may support the native treatment to embedded SQL statements. The native treatment allows a developer to employ existing tools (the ones used for the host programming language) for embedded SQL code.

Training and awareness: The role of focused training sessions to increase awareness of database quality among developers cannot be denied. Such sessions would enable them to learn from existing peer knowledge and keep themselves updated with the changing technology.

Database standards: Standards are a collection of common practices followed globally or within an organization to ensure the consistency and effectiveness of the database environment. A database element naming convention is an example of such a standard. Organizations may adopt stringent standards for designing database schemas to ensure the quality of the database system. Across the industry, a move toward stricter and comprehensive standards would prohibit some of the smells we identified.

Database APIs: Database APIs can also be improved to support high quality schema design. Apart from deprecating obsolete features and issuing a warning for common



mistakes, APIs may offer a new mechanism to verify the schema design. For example, a new CHECK statement (or an optional clause) may allow interested developers to check their schema design upfront and refactor the detected smells before they make their way to the production code.

5.5 Threats to Validity

This section presents threats to validity for all the experiments presented in this thesis.

5.5.1 Construct Validity

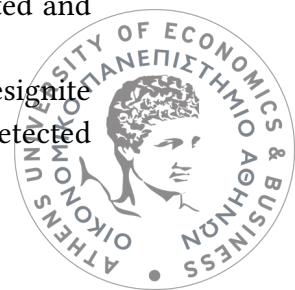
Construct validity measures the degree to which our tools and metrics actually measure the properties they are suppose to measure. It concerns the appropriateness of observations and inferences made on the basis of measurements taken during the study.

Static code analysis is typically prone to false-positives and false-negatives. To mitigate this concern, we employed a comprehensive set of tests for the tools presented in this thesis (viz. Designite, Puppeteer, and DbDeo) to rule out obvious deficiencies. Also, the effect of false positives and negatives is reduced when two or more streams of results are compared as in this experiment. Additionally, we found the results of manual validation of the detected instances by the tools very satisfactory.

The tools that we used in this thesis use various metric thresholds to detect smells. Although, some authors such as Rosenberg *et al.* [RSG99] have suggested the chosen threshold values after careful analysis. However, it is a known and accepted fact that there is no one globally accepted threshold set for various metrics [KKS⁺14, FBA11]. We chose the thresholds that are commonly used by the software engineering community. Moreover, in the specific case of Designite, the tool allows to change the thresholds if one would like to repeat the experiment with custom metric thresholds.

Further, a source-code analysis may adopt one of the numerous techniques to collect source-code information such as AST parsing, reflection, and string matching [Spi15]. For our tool Puppeteer, due to the lack of available parser library for Puppet, our tool uses regular expression based string matching extensively which is not as efficient as AST parsing. In addition to test the tool with unit-tests that check the correctness of the used regular expressions, we carried out manual testing to ensure the behaviour of the used expressions. Similarly, due to the lack of an available tool to extract cleansed SQL statements from a host source code, we implemented the extraction functionality in our tool (DbDeo) using regular expressions. Although, the regular expression-based solution cannot be as efficient as AST parsing (for example, separating SQL statements that are appearing in comments is inherently difficult with regular expressions). We employed two-step extraction process to overcome the deficiency. Additionally, we checked the results using both automated and manual tests.

In the context of using deep learning techniques for smell detection, we use Designite and DesigniteJava to detect smells in C# and Java code respectively and used the detected



smells by them as ground truth. Relying on the outcome of two different tools may pose a threat to validity especially in the case of transfer-learning. To mitigate the risk, we make sure that both the tools use exactly the same set of metrics and heuristics to detect smells. Also, we ensure the smell detection similarity by employing automated as well as manual testing. Similarly, to address potential threats posed by representational discrepancies between the two languages we ensure that Tokenizer generates same tokens for same or similar language constructs. For instance, all the common reserved words are mapped to the same integer token for both the programming languages.

5.5.2 Internal Validity

Internal validity refers to the validity of the research findings. It is primarily concerned with controlling the extraneous variables and outside influences that may impact the outcome.

In the context of refactoring simulation that we carried out in one of our experiments, there are many refactoring techniques to refactor the identified design smells. It is not feasible to predict precise impact of design smell refactorings on architecture smells. The observations of the refactoring simulations can vary in practice depending on the actual chosen refactoring. We follow a simple rationale to simulate the influence of design smell refactoring.

The higher the abstraction, the more important becomes the context of the system. Context and domain knowledge play an important role while detecting and refactoring, especially, design and architecture smells. Given the sheer scale, it was not possible to carry out a qualitative analysis for all the repositories. Considering the large number of repositories mined in the exploration, we believe that the results are still relevant and generalizable.

In the context of our investigation exploring the feasibility of applying transfer-learning for smell detection, we assume that both the programming languages are similar by paradigm, structure, and language constructs. It would be interesting to observe how two completely different programming languages (for example, Java and Python) can be combined in a transfer-learning experiment.

5.5.3 External Validity

External validity concerns generalizability and repeatability of the produced results. In our production code analysis study, we analyze only open-source C# repositories as subject systems. Given the fact that most of the current literature focuses only on subject systems written in Java programming language, our study complements the existing literature. Furthermore, we have considered a large set of 3,209 C# repositories of varied size and contexts, making this the largest mining study by scale so far for software smells.

To encourage the replication and building over the deep learning work, we have made all the tools, scripts, and data available online.¹

¹<https://github.com/tushartushar/DeepLearningSmells>



Similarly, our configuration code analysis experiment analyzes only Puppet repositories whereas there are many other configuration management systems. Although the employed tools are specific to one configuration language, the proposed theoretical model is general and language agnostic. We believe that it will open doors for similar studies for other configuration management systems.

For our database schema quality analysis, we cover syntaxes used for major database providers and new syntaxes can be adopted by modifying the currently used regular expressions. Also, the experiment is reproducible; we have made the tool open-source under a liberal license. Further, the raw data generated by the presented analysis has been made available online.

Finally, the extraction of the full schema of a database is not guaranteed using our employed method in our database quality assessment study. The implication of such a limitation is that our smell detection method will not report smells that may exist in the uncovered SQL statements.



Chapter 6

Conclusions and Future Work

| *Each conclusion marks a beginning.*

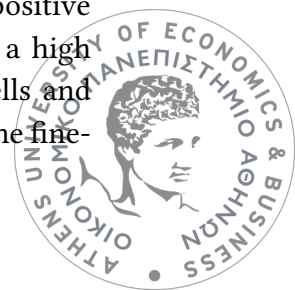
This thesis presents maintainability analysis on production source code and extends the scope of analysis to sub-domains of software systems. In this chapter, we summarize the results of our research, present the contributions of the thesis, elaborate our vision for future work, and conclude the thesis.

6.1 Summary of the Results

In the pursuit to perform a comprehensive maintainability analysis of production code written in C#, we perform a large-scale empirical study. We mine seven architecture, 19 design, 11 implementation smells from a large set of 3 209 open-source repositories containing more than 83 million lines of code. Stringing finer-grain code smells with the coarse-grain smells could make the task of maintaining high quality of a software product easier. Apart from exploring basic characteristics of smells (such as frequency of smells) arising at different granularities, we carry out correlation and collocation analysis also to identify the degree of relationships among smells at different granularities.

We find that *cyclic dependency*, *unutilized abstraction*, and *magic number* are the most frequently occurring architecture, design, and implementation smells respectively. This may prompt developers to pay additional attention to avoid frequently occurring smells. Our analysis observes that smell density and lines of code in a C# project do not show a strong correlation.

The co-occurrence analysis shows that the architecture smells exhibit a strong positive correlation ($\rho = 0.72$) with design smells. This implies that a project containing a high number of design smells would also exhibit a higher number of architecture smells and vice-versa. We perform fine-grain correlation also between individual smell pairs. The fine-



grain correlation analysis suggests that both the kinds of smells are not correlated and do not follow a monotonic relationship.

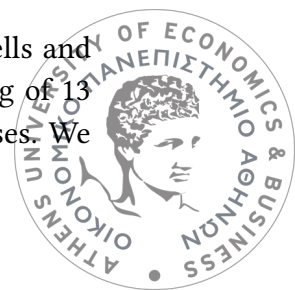
The collocation analysis reveals that unutilized abstraction and feature concentration are highly collocated. Similarly, cyclically-dependent modularization show relatively high collocation with cyclic dependency, scattered functionality, and dense structure architecture smells. Apart from the above-mentioned smell pairs, individual pairs of architecture and design smells do not collocate with each other. We also explore the potential influence of design smells refactoring on architecture smells. Our analysis reveals that upto one third of architecture smells (in case of *god component*) may get removed if we refactor all detected design smells. However, a significant number of architecture smells persist even after all the smells at design granularity were refactored. This observation emphasizes the need to carry out smell detection and refactoring for each granularity.

In our exploration with deep learning techniques to identify smells, we establish that deep learning methods can be used for smell detection. Specifically, we found that CNN and RNN deep learning models can be used for smell detection though with varying performance. We did not find a clearly superior method between 1D and 2D convolution neural networks; CNN-1D performed slightly better for the smells *empty catch block* and *multifaceted abstraction*, while CNN-2D performed superior than its one dimensional counterpart for *complex method* and *magic number*. Further, our results indicate that RNN performs far better than convolutional networks for smells *empty catch block* and *magic number*. Our experiment on applying transfer-learning proves the feasibility of practicing transfer-learning in the context of smell detection, especially for the implementation smells.

We extended the maintainability analysis to configuration code. We propose a catalog of 13 implementation and 11 design configuration smells based on commonly known best practices. We analyzed 4,621 Puppet repositories containing 142,662 Puppet files and more than 8.9 million lines of code using Puppeteer — a configuration smell detection tool that we developed. We investigated four research questions using smell instances detected by our analysis.

Our analysis found that the developers of Puppet repositories either do not introduce code-clones at all or they do it in a massive scale. Configuration smells belonging to a smell category tend to co-occur with configuration smells belonging to another smell category when correlation is computed by volume of identified smells. Design configuration smells show 9% higher average co-occurrence among themselves than the implementation configuration smells. This observation affirms the belief that one wrong or non-optimal design decision introduces many quality issues and therefore suggests the developers to take design decisions critically and diligently. Design configuration smell density shows negative correlation whereas implementation configuration smell density exhibits no correlation with size of a project. It shows that design configuration smells decrease as the size of the configuration code increases.

Further, we carried out a comparative study of relational database schema smells and its relationship with application and database characteristics. We present a catalog of 13 database schema smells based on commonly known best practices to design databases. We



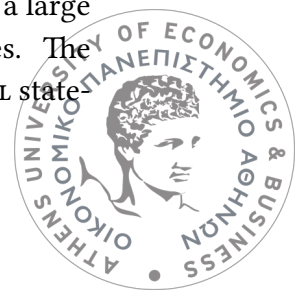
carried out a survey to understand developers perspective on database schema smells. We downloaded 16,052 open-source and acquired 840 industrial repositories, selected total 2925 repositories containing SQL statements, analyzed more than 629 million lines of code, extracted more than 393 thousand SQL statements, and detected more than 66 thousand instances of database schema smells. We investigated four research questions and provided empirical observations based on the data obtained.

We observed that the smell *index abuse* occurs most frequently in database code. We also found that some smells such as *adjacency list* show significantly higher proneness to occur in industrial projects compared to open-source projects. Our analysis shows that the size of the host application has no impact on the density of database smells; however, smell density shows positive correlation with the size of the database whereas application type (*Desktop*, *Mobile*, or *Web*) has no significant impact on database smell density. Another observation is that the use of an ORM framework does not avoid database schema smells. Finally, the smell *clone table* in industrial projects and smell *values in attribute definition* in open-source projects exhibit the highest co-occurrence with other database smells.

6.2 Contributions of the Thesis

In this section, we summarize the contributions offered by the thesis in two dimensions — *research* and *practice*. From the research perspective, we identify the following contributions from the thesis.

- A method to carry out large-scale empirical study (both in terms of number of subject systems and number of code smells detected) for production source code to understand characteristics of code smells at different granularities and to explore interesting relationships such as correlation and collocation.
- The software engineering research community may utilize the dataset of smells. The dataset could be useful in many ways including benchmarking and comparison as well as exploring other dimensions and characteristics of source code with smells.
- A method to identify smell catalog belonging to a domain (for instance, Infrastructure as Code). The method performs empirical study to analyze configuration code and explore characteristics specific to configuration code as well as analysis such as intra- as well as inter-category correlation.
- A way to design a qualitative survey aimed to collect developers' perspective on database schema design practices and smells.
- A method to investigate code quality of embedded SQL statements by mining a large set of repositories belonging to both open-source and proprietary categories. The method also outlines the challenges involved (such as extracting embedded SQL statements).

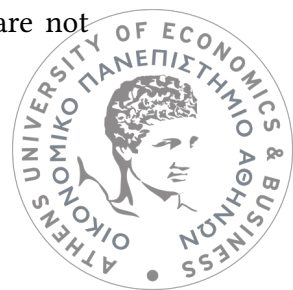


- A detailed mechanism to show the feasibility of detecting code smells using deep learning methods. Also, the method implements transfer-learning to showcase that a deep learning classifier trained from a programming language can be used to identify smelly code fragments belonging to another programming language.

Apart from research-oriented contributions, the thesis also offers contributions towards software engineering practice.

- The thesis offers a comprehensive code smells detection tool – Designite which could be used to detect a wide variety of implementation, design, and architecture smells in C# source code. Practitioners may use the tool and various features offered by the tool to identify maintainability issues in their code and reduce technical debt.
- As a by-product of our literature survey, we offer a large catalog of software smells. This catalog summarizes the smells by providing a description, related smells, tools that could be used to detect them, as well as the reference to the article where it was introduced. Not only the catalog is publicly accessible¹ but also the data and source code to build the smell catalog has been made open source.
- Practitioners may take advantage of correlation analysis that makes it imperative for the development teams to analyze and refactor smells at all granularities. Similarly, software development teams may emphasize the importance of detecting and refactoring smells at all granularities following our observations that a significant amount of architecture smells persists even if all the detected design smells were refactored.
- Practitioners can identify configuration smells using the tool viz. Puppeteer employed in this study and adopt best practices to write maintainable configuration code. Puppeteer [Sha19e] has been made open-source under liberal license and the time of writing this statement, it attracted 34 stars and 10 forks.
- Practitioners can learn the potential quality issues that may arise in their database schema so that they can avoid them. Furthermore, practitioners can identify database schema smells using our open-source tool (*i.e.*, DbDeo [Sha18b]) employed in this thesis. Finally, our results pinpoint areas where improvements in database APIs, tool support, training, and standards can increase the quality of database schemas.
- The tool developers may induct the deep learning methods in their smell detection tools for effective smell detection and using transfer-learning to detect smells for programming languages where the comprehensive code smell detection tools are not available.

¹<http://www.tusharma.in/smells/>



6.3 Future Work

We would like to extend and build upon our work presented in the thesis. Specifically, we would like to explore the following in the future.

- **Making code smells detection tools more effective:** The present set of smell detection tools generate a list of smells; the number of reported smells could be overwhelming for a large project. Despite some research attempts to prioritize smells, the production quality tools at large lack the features to identify smells that the developers really consider quality issues and propose highly impactful refactoring to eliminate them. Deep learning methods show a promising mechanism to achieve the goal.
- **Automated refactoring support for architecture smells:** Researchers have identified the lack of adequate tool support as one of the deterrents for software developers in adopting and performing refactoring regularly. Providing automated refactoring support to remove the detected code smells is inherently challenging even for implementation granularity given the various possible alternative refactorings for a smell. The challenge becomes immense when it comes to refactor architecture smells because a composite refactoring involves many smaller scale refactorings spanning multiple components.
- **Software data analytics:** Today's software systems are producing different kinds of data throughout the life-cycle; it includes, source-code itself and version control system data, reported bugs/issues and the discussion that follows, software quality data including metrics and smells, profiling data, logs and crash reports, and test execution data. All of this tells something about a different aspect about the software. Furthermore, combining them together may reveal further insights which can be actionable and useful for software development teams. For example, carrying out a postmortem analysis upon filing a new bug/issue may reveal interesting insights and patterns about the individual developers by analyzing code that has been changed, version control system data and test coverage. We would be very interested to apply exploratory machine learning methods to bring it to life.



Appendix I: Smell Definitions

Many authors have defined smells from their perspective. This appendix attempts to provide a consolidated list of such definitions.

1. Smells are certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring [Fow99].
2. Code smells are a metaphor to describe patterns that are generally associated with bad design and bad programming practices [vEM12].
3. Code smells are indicators or symptoms of the possible presence of design smells [MG07].
4. Code smells are implementation structures that negatively affect system lifecycle properties, such as understandability, testability, extensibility, and reusability; that is, code smells ultimately result in maintainability problems [GPEM09].
5. A “bad smell” describes a situation where there are hints that suggest there can be a design problem [PC09].
6. We define design defects as solutions to recurring problems that generate negative consequences on the quality of object-oriented systems [MGDM10].
7. Antipatterns are “poor” solutions to recurring implementation and design problems that impede the maintenance and evolution of programs [KVGs11].
8. Anti-patterns are bad solutions to recurring design problems [FBA11].
9. An anti-pattern is a commonly occurring solution to a recurring problem that will typically negatively impact code quality. Code smells are considered to be symptoms of anti-patterns and occur at source code level [PZ12].
10. Antipatterns are defined as patterns that appear obvious but are ineffective or far from optimal in practice, representing worst practices about how to structure and design an ontology [RCSZ⁺12].
11. Anti-patterns are “poor” solutions to recurring design and implementation problems [MAB⁺12b].

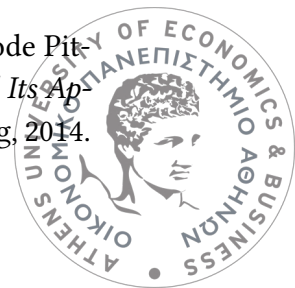


12. Developers often introduce bad solutions, anti-patterns, to recurring design problems in their systems and these anti-patterns lead to negative effects on code quality [JGHK13].
13. Linguistic antipatterns in software systems are recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity, thus possibly impairing program understanding [ADPAG13].
14. Design smells are structures in the design that indicate violation of fundamental design principles and negatively impact design quality [SSS14].
15. Code smells are indicators of deeper design problems that may cause difficulties in the evolution of a software system [Yam14].
16. Performance Antipatterns define bad practices that induce performance problems, and their solutions [CDMT14].
17. Antipatterns are typically a commonly used set of design and coding constructs which might appear intuitive initially, but eventually may be detrimental to one or more aspects of the system [SA14].
18. Bad design practices at the code level are known as bad smells in the literature [KEA16].
19. Code smells — microstructures in the program — have been used to reveal surface indications of a design problem [dSS16].
20. Configuration smells are the characteristics of a configuration program or script that violate the recommended best practices and potentially affect the program's quality in a negative way [SFS16].



Bibliography

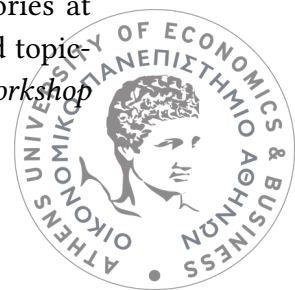
- [AAK⁺17] Osama Abdeljaber, Onur Avci, Serkan Kiranyaz, Moncef Gabbouj, and Daniel J Inman. Real-time vibration-based structural damage detection using one-dimensional convolutional neural networks. *Journal of Sound and Vibration*, 388:154–170, 2017.
- [ABDS18] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.
- [ABT15] Davide Arcelli, Luca Berardinelli, and Catia Trubiani. Performance Antipattern Detection through fUML Model Library. In *WOSP '15: Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development*, pages 23–28. University of L'Aquila, ACM, January 2015.
- [ACSS15] Diogo Almeida, José Creissac Campos, João Saraiva, and João Carlos Silva. Towards a catalog of usability smells. In *SAC '15: Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 175–181. University of Minho, ACM, April 2015.
- [AD15] Jehad Al Dallah. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology*, 58:231–249, January 2015.
- [ADPAG13] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. A New Family of Software Anti-patterns: Linguistic Anti-patterns. In *CSMR '13: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, pages 187–196. IEEE Computer Society, March 2013.
- [AFBZ12] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *The Journal of Object Technology*, 11(2):5:1–38, 2012.
- [AFF14] Péricles Alves, Eduardo Figueiredo, and Fabiano Ferrari. Avoiding Code Pitfalls in Aspect-Oriented Programming. In *Computational Science and Its Applications – ICCSA 2012*, pages 31–46. Springer International Publishing, 2014.



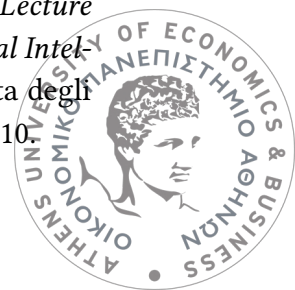
- [AGJ08] Silvia T Acuña, Marta Gómez, and Natalia Juristo. Towards understanding the relationship between team climate and software quality—a quasi-experimental study. *Empirical Software Engineering*, 13(4):339–342, August 2008.
- [AHTM11] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. The effect of lexicon bad smells on concept location in source code. In *Proceedings - 11th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2011*, pages 125–134. Fondazione Bruno Kessler, Trento, Italy, IEEE, November 2011.
- [APFC15] Ramon Abílio, Juliana Padilha, Eduardo Figueiredo, and Heitor Costa. Detecting Code Smells in Software Product Lines – An Exploratory Study. In *ITNG '15: Proceedings of the 2015 12th International Conference on Information Technology - New Generations*, pages 433–438. IEEE Computer Society, April 2015.
- [APG17] Carol V Alexandru, Sebastiano Panichella, and Harald C Gall. Replicating parser behavior using neural machine translation. In *Proceedings of the 25th International Conference on Program Comprehension*, pages 316–319. IEEE Press, 2017.
- [APS16] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.
- [Bai94] Kenneth D Bailey. *Typologies and taxonomies: an introduction to classification techniques*, volume 102. Sage, 1994.
- [BBEAM10] Sérgio Bryton, Fernando Brito E Abreu, and Miguel Monteiro. Reducing subjectivity in code smells detection: Experimenting with the Long Method. In *Proceedings - 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010*, pages 337–342. Faculdade de Ciencias e Tecnologia, New University of Lisbon, Caparica, Portugal, IEEE, December 2010.
- [BCV13] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.
- [BDLDP⁺15] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, January 2015.



- [Bec02] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [BGH⁺08] David Binkley, Nicolas Gold, Mark Harman, Zheng Li, Kiarash Mahdavi, and Joachim Wegener. Dependence Anti Patterns. In *Aramis 2008 - 1st International Workshop on Automated engineering of Autonomous and runtime evolving Systems, and ASE2008 the 23rd IEEE/ACM Int. Conf. Automated Software Engineering*, pages 25–34. King’s College London, London, United Kingdom, IEEE, December 2008.
- [BGvS11] Isela Macia Bertran, Alessandro Garcia, and Arndt von Staa. An exploratory study of code smells in evolving aspect-oriented systems. In *AOSD ’11: Proceedings of the tenth international conference on Aspect-oriented software development*, page 203. Pontifical Catholic University of Rio de Janeiro, ACM, March 2011.
- [BINF12] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *34th International Conference on Software Engineering (ICSE)*, pages 419–429, June 2012.
- [BKG19] Antoine Barbez, Foutse Khomh, and Yann-Gaël Guéhéneuc. A machine-learning based ensemble method for anti-patterns detection, 2019.
- [BLV07] Huib Van Den Brink, Rob Van Der Leek, and Joost Visser. Quality assessment for embedded sql. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM ’07*, pages 163–170. IEEE Computer Society, 2007.
- [BMMM98] William H. Brown, Raphael C. Malveau, Hays W. ”Skip” McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., 1st edition, 1998.
- [BMR⁺96a] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1 edition, 1996.
- [BMR⁺96b] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1 edition, 1996.
- [BPD17] Christos Baziotis, Nikos Pelekis, and Christos Doulkeridis. Datastories at semeval-2017 task 4: Deep lstm with attention for message-level and topic-based sentiment analysis. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 747–754, 2017.



- [BQO⁺12] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *IEEE International Conference on Software Maintenance, ICSM*, pages 56–65. Universita di Salerno, Salerno, Italy, IEEE, December 2012.
- [BQO⁺14] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? An empirical study. *Empirical Software Engineering*, 20(4):1052–1094, May 2014.
- [BSH⁺11] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11*, pages 53–62. ACM, 2011.
- [CDMT14] Vittorio Cortellessa, Antinisca Di Marco, and Catia Trubiani. An approach for modeling and detecting software performance antipatterns based on first-order logics. *Software and Systems Modeling (SoSyM)*, 13(1):391–432, February 2014.
- [Che15] T. Chen. Improving the quality of large-scale database-centric software systems by analyzing database access code. *2015 31st IEEE International Conference on Data Engineering Workshops (ICDEW)*, 00, 2015.
- [Che18] Chef: Do Change, Last accessed on: Nov 14, 2018. Available at: <https://www.chef.io/>.
- [Cho17] Francois Chollet. *Deep learning with python*. Manning Publications Co., 2017.
- [CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transaction of Software Engineering*, 20(6):476–493, June 1994.
- [CMC15] Gabriela Czibula, Zsuzsanna Marian, and Istvan Gergely Czibula. Detecting software design defects using relational association rule mining. *Knowledge and Information Systems*, 42(3):545–577, March 2015.
- [CMRP16] Karina Curcio, Andreia Malucelli, Sheila Reinehr, and Marco Antônio Paludo. An analysis of the factors determining software product quality: A comparative study. *Computer Standards & Interfaces*, 48:10–18, November 2016.
- [CMRT10] Vittorio Cortellessa, Anne Martens, Ralf Reussner, and Catia Trubiani. A process to effectively identify “guilty” performance antipatterns. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 368–382. Universita degli Studi dell’Aquila, L’Aquila, Italy, Springer Berlin Heidelberg, April 2010.



- [Coh60] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.
- [CPD16] PMD-CPD: Copy Paste Detector. <https://pmd.github.io/>, 2016. [Online; accessed 22-Jan-2016].
- [CS78] E.F. Connor and D. Simberloff. Species number and compositional similarity of the galapagos flora and avifauna. *Ecological Monographs*, (48):219–248, 1978.
- [CSJ⁺14] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *ICSE 2014: Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1001–1012. Queen’s University, Kingston, ACM, May 2014.
- [CvMG⁺14] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.
- [dAAC14a] Hugo Sica de Andrade, Eduardo Almeida, and Ivica Crnkovic. Architectural bad smells in software product lines: An exploratory study. In *Proceedings of the WICSA 2014 Companion Volume*, WICSA ’14 Companion, pages 12:1–12:6. ACM, 2014.
- [dAAC14b] Hugo Sica de Andrade, Eduardo Almeida, and Ivica Crnkovic. Architectural bad smells in software product lines: An exploratory study. In *Proceedings of the WICSA 2014 Companion Volume*, WICSA ’14 Companion, pages 12:1–12:6. ACM, 2014.
- [dbS10] What are the most common SQL anti-patterns? <http://stackoverflow.com/questions/346659/what-are-the-most-common-sql-anti-patterns>, 2010. [Online; accessed 25-Jan-2017].
- [DD16] Tuhin Kanti Das and Juergen Dingel. Model development guidelines for UML-RT: conventions, patterns and antipatterns. *Software & Systems Modeling*, pages 1–36, July 2016.
- [DDS⁺09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.
- [Deu01] Deursen, A. Van and L. Moonen and Bergh, A. Van Den and G. Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International*



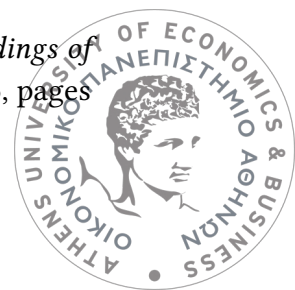
- Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.
- [DPXT13] Jiang Dexun, Ma Peijun, Su Xiaohong, and Wang Tiantian. Detection and Refactoring of Bad Smell Caused by Large Scale. *International Journal of Software Engineering & Applications*, 4(5):1–13, September 2013.
- [dSS16] Leonardo da Silva Sousa. Spotting design problems with smell agglomerations. In *ICSE '16: Proceedings of the 38th International Conference on Software Engineering Companion*, pages 863–866. Pontifical Catholic University of Rio de Janeiro, ACM, May 2016.
- [EAM09] Mohamed El-Attar and James Miller. Improving the quality of use case models using antipatterns. *Software & Systems Modeling*, 9(2):141–160, February 2009.
- [Ern17] Michael D Ernst. Natural language is a programming language: Applying natural language processing to software development. In *LIPICs-Leibniz International Proceedings in Informatics*, volume 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- [EV15] Erki Eessaar and Janina Voronova. *Using SQL Queries to Evaluate the Design of SQL Databases*, pages 179–186. Springer International Publishing, 2015.
- [FBA11] Rahma Fourati, Nadia Bouassida, and Hanène Ben Abdallah. A Metric-Based Approach for Anti-pattern Detection in UML Designs. In *Computer and Information Science 2011*, pages 17–33. Springer Berlin Heidelberg, 2011.
- [FBB⁺12] Kecia A M Ferreira, Mariza A S Bigonha, Roberto S Bigonha, Luiz F O Mendes, and Heitor C Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257, February 2012.
- [FDW⁺16] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 609–613. IEEE, 2016.
- [FFM⁺13] Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bartosz Walter, and Pawel Martenka. Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains. In *2013 IEEE International Conference on Software Maintenance (ICSM)*, pages 260–269. IEEE, September 2013.
- [FFZY15] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. Automatic metric thresholds derivation for code smell detection.



- In *WETSoM '15: Proceedings of the Sixth International Workshop on Emerging Trends in Software Metrics*, pages 44–53. University of Lugano, IEEE Press, May 2015.
- [FGL12] Stephen R Foster, William G Griswold, and Sorin Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 222–232. IEEE, 2012.
- [FM13] Amin Milani Fard and Ali Mesbah. JSNOSE: Detecting javascript code smells. In *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013*, pages 116–125. The University of British Columbia, Vancouver, Canada, IEEE, January 2013.
- [FM17] Wei Fu and Tim Menzies. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 49–60. ACM, 2017.
- [FOV⁺16] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *EASE '16: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 18–12. Federal University of Minas Gerais, ACM, June 2016.
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley Professional, 1 edition, 1999.
- [Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 1 edition, 2002.
- [FPRZ16] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, and Marco Zanoni. Automatic detection of instability architectural smells. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 433–437. IEEE, 2016.
- [FS15] Shizhe Fu and Beijun Shen. Code Bad Smell Detection through Evolutionary Data Mining. In *International Symposium on Empirical Software Engineering and Measurement*, pages 41–49. Shanghai Jiaotong University, Shanghai, China, IEEE, November 2015.
- [FSMS15] Wolfram Fenske, Sandro Schulze, Daniel Meyer, and Gunter Saake. When code smells twice as much: Metric-based detection of variability-aware code smells. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings*, pages 171–180. Otto von Guericke University of Magdeburg, Magdeburg, Germany, IEEE, November 2015.



- [FTC07] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and Removal of Feature Envy Bad Smells. In *2007 IEEE International Conference on Software Maintenance*, pages 519–520. Panepistimion Makedonias, Thessaloniki, Greece, IEEE, 2007.
- [FVE91] Daniel J Felleman and David C Van Essen. Distributed hierarchical processing in the primate cerebral cortex. *Cerebral Cortex*, 1(1):1–47, 1991.
- [Gar14] Joshua Garcia. Technical report: Architectural Smell Definitions and Formalizations. <http://csse.usc.edu/TECHRPTS/2014/reports/usc-csse-2014-500.pdf>, 2014. [Online; accessed 16-June-2017].
- [GBCB16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.
- [GEBK15] Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini. On the use of design defect examples to detect model refactoring opportunities. *Software Quality Journal*, pages 1–19, March 2015.
- [GG15] Yarin Gal and Zoubin Ghahramani. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. *arXiv e-prints*, page arXiv:1512.05287, Dec 2015.
- [GGC14] E. Guimaraes, A. Garcia, and Y. Cai. Exploring blueprints on the prioritization of architecturally relevant code anomalies – a controlled experiment. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 344–353, July 2014.
- [Git16] GitHub. <https://github.com/>, 2016. [Online; accessed 22-Jan-2016].
- [GJM13] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 273–278. IEEE, 2013.
- [GKA⁺16] Latifa Guerrouj, Zeinab Kermansaravi, Venera Arnaoudova, Benjamin C M Fung, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Investigating the relation between lexical smells and change- and fault-proneness: an empirical study. *Software Quality Journal*, pages 1–30, May 2016.
- [GL16] Joseph Yossi Gil and Gal Lalouche. When do Software Complexity Metrics Mean Nothing? – When Examined out of Context. *The Journal of Object Technology*, 15(1):2:1, 2016.
- [Gou13] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13*, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.



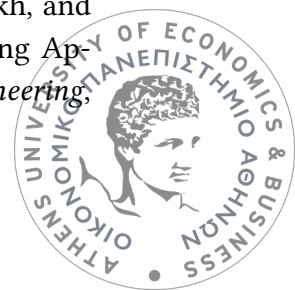
- [GPEM09] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, QoSA '09, pages 146–162. Springer-Verlag, 2009.
- [GPKS17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, pages 1345–1351, 2017.
- [GS12] G. Gousios and D. Spinellis. GHTorrent: Github’s data from a firehose. In *9th IEEE Working Conference on Mining Software Repositories*, pages 12–21, June 2012.
- [GSK⁺17] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2017.
- [GvDS13] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated Detection of Test Fixture Strategies and Smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*, pages 322–331. IEEE, January 2013.
- [GVG⁺] E. Guimarães, S. Vidal, A. Garcia, J. A. Diaz Pace, and C. Marcos. Exploring architecture blueprints for prioritizing critical code anomalies: Experiences and tool support. *Software: Practice and Experience*, 48(5):1077–1106.
- [HAT⁺04] H H Hallal, E Alikacem, W P Tunney, S Boroday, and A Petrenko. Antipattern-Based Detection of Deficiencies in Java Multithreaded Software. In *QSIC '04: Proceedings of the Quality Software, Fourth International Conference*, pages 258–267. Cent de Recherche Informatique de Montreal, IEEE Computer Society, September 2004.
- [HBS⁺12] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.
- [HD17] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773. ACM, 2017.
- [HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1 edition, 2010.



- [HJE⁺13] Benedikt Hauptmann, Maximilian Junker, Sebastian Eder, Lars Heinemann, Rudolf Vaas, and Peter Braun. Hunting for smells in natural language tests. In *ICSE '13: Proceedings of the 2013 International Conference on Software Engineering*, pages 1217–1220. Technical University of Munich, IEEE Press, May 2013.
- [HLZ16] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, pages 1606–1612, 2016.
- [HMR16] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. An empirical study of the performance impacts of Android code smells. In *MOBILESoft '16: Proceedings of the International Workshop on Mobile Software Engineering and Systems*. Universite Lille 2 Droit et Sante, ACM, May 2016.
- [HOT06] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.
- [HPvD12] F. Hermans, M. Pinzger, and A. van Deursen. Detecting code smells in spreadsheet formulas. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 409–418, Sept 2012.
- [HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [HW62] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.
- [HZBS14] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):33–39, September 2014.
- [IKCZ16] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2073–2083, 2016.
- [IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*, pages 448–456. JMLR. org, 2015.
- [JA15] Yujian Jiang and Bram Adams. Co-evolution of Infrastructure and Source Code: An Empirical Study. In *Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15*, pages 45–55, Piscataway, NJ, USA, 2015. IEEE Press.



- [JGHK13] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. Mining the relationship between anti-patterns dependencies and fault-proneness. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 351–360. Ecole Polytechnique de Montreal, Montreal, Canada, IEEE, 2013.
- [JZ15] Rie Johnson and Tong Zhang. Effective use of word order for text categorization with convolutional neural networks. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 103–112, 2015.
- [Kar10] Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf, 1st edition, 2010.
- [KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [KBF16] Oren Z Kraus, Jimmy Lei Ba, and Brendan J Frey. Classifying and segmenting microscopy images with deep multiple instance learning. *Bioinformatics*, 32(12):i52–i59, 2016.
- [KDPG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84. Ecole Polytechnique de Montreal, Montreal, Canada, IEEE, December 2009.
- [KDPGA12] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, June 2012.
- [KEA16] Yasser A Khan and Mohamed El-Attar. Using model transformation to refactor use case models based on antipatterns. *Information Systems Frontiers*, 18(1):171–204, 2016.
- [KH09] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.
- [KHRS12] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. Do static type systems improve the maintainability of software systems? An empirical study. In *2012 IEEE 20th International Conference on Program Comprehension (ICPC)*, pages 153–162. Universitat Duisburg-Essen, Essen, Germany, IEEE, 2012.
- [KKS⁺14] Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and Ali Ouni. A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection. *IEEE Transactions on Software Engineering*, 40(9):841–861, 2014.



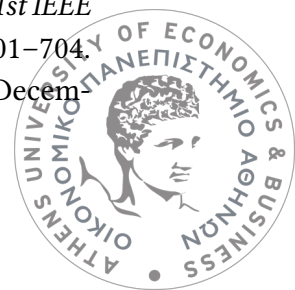
- [Koe95] Andrew Koenig. Patterns and antipatterns. *JOOOP*, 8(1):46–48, 1995.
- [KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [KVG09] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A Bayesian Approach for the Detection of Code and Design Smells. In *QSIC '09: Proceedings of the 2009 Ninth International Conference on Quality Software*, pages 305–314. IEEE Computer Society, August 2009.
- [KVG11] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. In *Journal of Systems and Software*, pages 559–572. Ecole Polytechnique de Montreal, Montreal, Canada, 2011.
- [KŽ07] Jaroslav Král and Michal Žemlička. The most important service-oriented antipatterns. In *2nd International Conference on Software Engineering Advances - ICSEA 2007*, pages 29–29. Charles University in Prague, Prague, Czech Republic, IEEE, December 2007.
- [Lar16a] Gary Larizza. Building a Functional Puppet Workflow Part 1: Module Structure. <http://www.webcitation.org/6g23RY7yS>, 2016. [Online; accessed 15-Mar-2016].
- [Lar16b] Gary Larizza. Building a Functional Puppet Workflow Part 2: Module Structure. <http://www.webcitation.org/6g23YeuFl>, 2016. [Online; accessed 15-Mar-2016].
- [Lar16c] Gary Larizza. Doing the Refactor Dance — Making Your Puppet Modules More Modular. <http://www.webcitation.org/6g23dnNKO>, 2016. [Online; accessed 15-Mar-2016].
- [LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [LBG⁺16] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 599–609, 2016.
- [LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.



- [LCB10] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *AT&T Labs [Online]*. Available: <http://yann.lecun.com/exdb/mnist>, 2, 2010.
- [LCCY13] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis. Identification of refused bequest code smells. In *2013 IEEE International Conference on Software Maintenance*, pages 392–395, Sept 2013.
- [LGTB97] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.
- [LHZL17] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pages 318–328. IEEE, 2017.
- [LK00] A Lauder and S Kent. Legacy System Anti-Patterns and a Pattern-Oriented Migration Response. In *Systems Engineering for Business Process Change*, pages 239–250. Springer London, 2000.
- [LLNL16] Hui Liu, Qiurong Liu, Zhendong Niu, and Yang Liu. Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection. *IEEE Transactions on Software Engineering*, 42(6):544–558, June 2016.
- [LLSM18] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural decay in open-source software. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 176–17609, April 2018.
- [LMSN12] 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*, 38(1):220–235, 2012.
- [Lon01] John Long. Software reuse antipatterns. *ACM SIGSOFT Software Engineering Notes*, 26(4):68–76, July 2001.
- [LPM15] Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, 2015.
- [LR06] Martin Lippert and Stephen Roock. *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006.
- [LR15] Mathieu Lavallée and Pierre N Robillard. Why good developers write bad code: An observational case study of the impacts of organizational factors



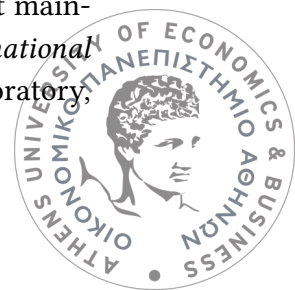
- on software quality. In *Proceedings - International Conference on Software Engineering*, pages 677–687. Polytechnique Montréal, Montreal, Canada, IEEE, August 2015.
- [LVKM⁺14] Mario Linares-Vásquez, Sam Klock, Collin McMillan, Aminata Sabané, Denys Poshyvanyk, and Yann-Gaël Guéhéneuc. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps. In *ICPC 2014: Proceedings of the 22nd International Conference on Program Comprehension*, pages 232–243. The College of William and Mary, ACM, June 2014.
- [LXZ18] Hui Liu, Zhifeng Xu, and Yanzhen Zou. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pages 385–396, New York, NY, USA, 2018. ACM.
- [LYC17] Song-Mi Lee, Sang Min Yoon, and Heeryon Cho. Human activity recognition from accelerometer data using convolutional neural network. In *Big Data and Smart Computing (BigComp), 2017 IEEE International Conference on*, pages 131–134. IEEE, 2017.
- [MAB⁺12a] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, and Esma Aïmeur. SMURF: A SVM-based incremental anti-pattern detection approach. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 466–475. Ptidej Team, IEEE, December 2012.
- [MAB⁺12b] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. Support vector machines for anti-pattern detection. In *ASE 2012: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 278–281. Polytechnic School of Montreal, ACM, September 2012.
- [Mar01] K. Marquardt. Dependency structures—architectural diagnoses and therapies. In *Proceedings of the EuroPLOp*, 2001.
- [Mar02] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.
- [Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance, ICSM '04*, pages 350–359. IEEE Computer Society, 2004.
- [Mar05] R Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 701–704. Universitatea Politehnica din Timisoara, Timisoara, Romania, IEEE, December 2005.



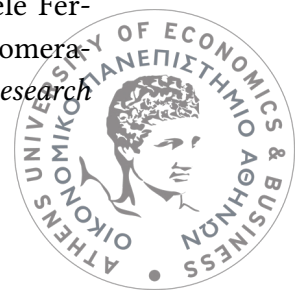
- [Mar10] James Martens. Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742, 2010.
- [MBC14] A. Martini, J. Bosch, and M. Chaudron. Architecture technical debt: Understanding causes and a qualitative model. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 85–92, Aug 2014.
- [MCKX15] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In *WICSA*, pages 51–60. IEEE Computer Society, 2015.
- [MDP⁺11] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.
- [MFBR18] Antonio Martini, Francesca Arcelli Fontana, Andrea Biaggi, and Riccardo Roveda. Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company. In *12th European Conference on Software Architecture (ECSA 2018)*, September 2018.
- [MG07] Naouel Moha and Yann-Gaël Guéhéneuc. Decor: a tool for the detection of design defects. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 527–528. University of Montreal, ACM, 2007.
- [MGDM10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.
- [MGP⁺12] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. Are automatically-detected code anomalies relevant to architectural modularity? In *the 11th annual international conference*, pages 167–178. ACM Press, 2012.
- [MGvS10] Isela Macia, Alessandro Garcia, and Arndt von Staa. Defining and applying detection strategies for aspect-oriented code smells. In *Proceedings - 24th Brazilian Symposium on Software Engineering, SBES 2010*, pages 60–69. Pontificia Universidade Catolica do Rio de Janeiro, Rio de Janeiro, Brazil, IEEE, December 2010.
- [MHB08] Emerson Murphy-Hill and Andrew P Black. Seven habits of a highly effective smell detector. In *the 2008 international workshop*, pages 36–40. Portland State University, Portland, United States, ACM Press, 2008.



- [MHB10] Emerson Murphy-Hill and Andrew P Black. An interactive ambient visualization for code smells. In *SOFTVIS '10: Proceedings of the 5th international symposium on Software visualization*. North Carolina State University, ACM, October 2010.
- [MKCN17] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, Dec 2017.
- [MKMD16] Usman Mansoor, Marouane Kessentini, Bruce R Maxim, and Kalyanmoy Deb. Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal*, pages 1–24, February 2016.
- [ML06] Mika V Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, September 2006.
- [MLZ⁺16] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, volume 2, page 4, 2016.
- [MNK⁺02] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, page 87. IEEE Computer Society, June 2002.
- [MS16] Alan MacCormack and Daniel J. Sturtevant. Technical debt and system architecture: The impact of coupling on defect-related activity. *Journal of Systems and Software*, 120:170 – 182, 2016.
- [MT04] Tom Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.
- [Mun05] Matthew James Munro. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 15–15. University of Strathclyde, IEEE Computer Society, September 2005.
- [MVL03] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, September 2003.
- [MY12] Leon Moonen and Aiko Yamashita. Do code smells reflect important maintainability aspects? In *ICSM '12: Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*. Simula Research Laboratory, IEEE Computer Society, September 2012.



- [NC15] Csaba Nagy and Anthony Cleve. Mining stack overflow for discovering error patterns in sql queries. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 00:516–520, 2015.
- [NNN⁺12] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. Detection of embedded code smells in dynamic web applications. In *ASE 2012: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 282–285. Iowa State University, ACM, September 2012.
- [NNN13] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 651–654. ACM, 2013.
- [Non15] Kwankamol Nongpong. Feature envy factor: A metric for automatic feature envy detection. In *Proceedings of the 2015-7th International Conference on Knowledge and Smart Technology, KST 2015*, pages 7–12. Assumption University, Bangkok, Bangkok, Thailand, IEEE, January 2015.
- [NPT⁺18] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 00, pages 612–621, March 2018.
- [OAH⁺18] Jordan Ott, Abigail Atchison, Paul Harnack, Natalie Best, Haley Anderson, Cristiano Firmani, and Erik Linstead. Learning lexical features of programming languages from imagery using convolutional neural networks. pages 336–339, 2018.
- [OCBZ09] Steffen Olbrich, Daniela S Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 390–400. IEEE, August 2009.
- [OFN⁺15] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 574–584. IEEE, 2015.
- [OGC⁺15] Willian N. Oizumi, Alessandro F. Garcia, Thelma E. Colanzi, Manuele Ferreira, and Arndt V. Staa. On the relationship of code-anomaly agglomerations and architectural problems. *Journal of Software Engineering Research and Development*, 3(1):11, 2015.



- [OGdSS⁺16] Willian Oizumi, Alessandro Garcia, Leonardo da Silva Sousa, Bruno Cafeo, and Yixue Zhao. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 440–451, 2016.
- [OKAG10] Rocco Oliveto, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Numerical Signatures of Antipatterns: An Approach Based on B-Splines. In *CSMR '10: Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, pages 248–251. IEEE Computer Society, March 2010.
- [OKKI15] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, and Katsuro Inoue. Web Service Antipatterns Detection Using Genetic Programming. In *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1351–1358. Osaka University, ACM, July 2015.
- [Pal18] A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology*, 99:1 – 10, 2018.
- [PBDP⁺15] Fabio Palomba, 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(5):462–489, May 2015.
- [PBP⁺14] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110. IEEE, July 2014.
- [PC09] Javier Pérez and Yania Crespo. Perspectives on automated correction of bad smells. In *the joint international and annual ERCIM workshops*, pages 99–108. Universidad de Valladolid, Valladolid, Spain, ACM Press, 2009.
- [PDLBO14] Fabio Palomba, Andrea De Lucia, Gabriele Bavota, and Rocco Oliveto. Anti-Pattern Detection. In *Anti-pattern detection: Methods, challenges, and open issues*, pages 201–238. Elsevier, 2014.
- [PDMG14] Francis Palma, Johann Dubois, Naouel Moha, and Yann-Gaël Guéhéneuc. Detection of REST patterns and antipatterns: A heuristics-based approach. In Xavier Franch, Aditya K Ghose, Grace A Lewis, and Sami Bhiri, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 230–244. Université du Québec à Montreal, Montreal, Canada, Springer Berlin Heidelberg, January 2014.



- [PHN⁺15] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. In *International Conference on Machine Learning*, pages 1093–1102, 2015.
- [PM15] Francis Palma and Naouel Mohay. A study on the taxonomy of service antipatterns. In *2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-Patterns Prevention, PPAP 2015 - Proceedings*, pages 5–8. Ecole Polytechnique de Montreal, Montreal, Canada, IEEE, January 2015.
- [PMG13] Francis Palma, Naouel Moha, and Yann-Gaël Guéhéneuc. Detection of process antipatterns: A BPEL perspective. In *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, pages 173–177. École Polytechnique, Canada, IEEE, January 2013.
- [PNSLB16] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 39–40. ACM, 2016.
- [PNT⁺15] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Landfill: An open dataset of code smells with public evaluation. In *Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15*, pages 482–485. IEEE Press, 2015.
- [PPDL⁺16] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. A textual-based technique for Smell Detection. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. Università di Salerno, Salerno, Italy, IEEE, 2016.
- [PPF⁺14] Juliana Padilha, Juliana Pereira, Eduardo Figueiredo, Jussara Almeida, Alessandro Garcia, and Claudio Sant’Anna. On the effectiveness of concern metrics to detect code smells: An empirical study. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 656–671. Universidade Federal de Minas Gerais, Belo Horizonte, Brazil, Springer International Publishing, January 2014.
- [PR11] Mikhail Perepletchikov and Caspar Ryan. A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software. *IEEE Transactions on Software Engineering*, 37(4):449–465, August 2011.
- [Pup16a] Sonar Puppet. SonarQube Puppet Plugin, Last accessed on: 22nd Jan 2016. Available at: <https://github.com/iwarapter/sonar-puppet>.



- [Pup16b] Puppet Forge: a repository of Puppet modules, Last accessed on: 22nd Jan 2016. Available at: <https://forge.puppetlabs.com>.
- [Pup16c] Puppet-lint: Puppet code style checker, Last accessed on: 22nd Jan 2016. Available at: <http://puppet-lint.com>.
- [Pup18] Puppet: Deliver better software, faster, Last accessed on: Nov 14, 2018. Available at: <https://puppet.com/>.
- [PVZ⁺15] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, et al. Deep face recognition. In *BMVC*, volume 1, page 6, 2015.
- [PZ12] Ralph Peters and Andy Zaidman. Evaluating the lifespan of code smells using software repository mining. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 411–416. IEEE Computer Society, 2012.
- [RA15] Ghulam Rasool and Zeeshan Arshad. A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895, November 2015.
- [Ram10] Girish Maskeri Rama. A desiderata for refactoring-based software modularity improvement. In *ISEC '10: Proceedings of the 3rd India software engineering conference*, pages 93–102. Infosys Technologies Limited India, ACM, February 2010.
- [RCSZ⁺12] Catherine Roussey, Oscar Corcho, Ondrej Svab-Zamazal, François Scharffe, and Stephan Bernard. SPARQL-DL queries for antipattern detection. In *WOP'12: Proceedings of the 3rd International Conference on Ontology Patterns - Volume 929*, pages 85–96. Cemagref, CEUR-WS.org, November 2012.
- [Red17] Redgate. 119 SQL Code Smells. <http://assets.red-gate.com/community/books/sql-code-smells.pdf>, 2017. [Online; accessed 8-Feb-2017].
- [RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.
- [Rie96] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1 edition, 1996.
- [Rob10] Gregorio Robles. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 171–180. IEEE, 2010.
- [RSG99] Linda H. Rosenberg, Ruth Stapko, and Al Gallo. Risk-based object oriented testing. In *Twenty-Fourth Annual Software Engineering Workshop*, Greenbelt, MD, December 1999. NASA, Software Engineering Laboratory.



- [SA13] Vibhu Saujanya Sharma and Samit Anwer. Detecting Performance Antipatterns before Migrating to the Cloud. In *CLOUDCOM '13: Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01*, pages 148–151. IEEE Computer Society, December 2013.
- [SA14] Vibhu Saujanya Sharma and Samit Anwer. Performance antipatterns: Detection and evaluation of their effects in the cloud. In *Proceedings - 2014 IEEE International Conference on Services Computing, SCC 2014*, pages 758–765. Accenture Services Pvt Ltd., India, Bangalore, India, IEEE, January 2014.
- [SCY⁺16] Tsubasa Saika, Eunjong Choi, Norihiro Yoshida, Shusuke Haruna, and Katsuro Inoue. Do Developers Focus on Severe Code Smells? In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 1–3. IEEE, 2016.
- [SDPAG13] Aminata Sabané, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. A Study on the Relation between Antipatterns and the Cost of Class Unit Testing. In *CSMR '13: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, pages 167–176. IEEE Computer Society, March 2013.
- [SFS16] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Workshop on Mining Software Repositories, MSR'16*, pages 189–200, 2016.
- [Sha] title = Configuration smells dataset month = may year = 2016 doi = 10.5281/zenodo.2567067 url = <https://doi.org/10.5281/zenodo.2567067> Sharma, Tushar.
- [Sha16] Tushar Sharma. Designite - A Software Design Quality Assessment Tool, May 2016. <http://www.designite-tools.com>.
- [Sha18a] Tushar Sharma. Database schema quality analysis dataset, May 2018.
- [Sha18b] Tushar Sharma. Dbdeo: Database schema smells detector, May 2018. <https://github.com/tushartushar/dbdeo>.
- [Sha18c] Tushar Sharma. Designitejava, December 2018. <https://github.com/tushartushar/DesigniteJava>.
- [Sha19a] Tushar Sharma. Codesplit for c#, February 2019.
- [Sha19b] Tushar Sharma. Codesplitjava, February 2019. <https://github.com/tushartushar/CodeSplitJava>.
- [Sha19c] Tushar Sharma. A dataset of code smells, January 2019.



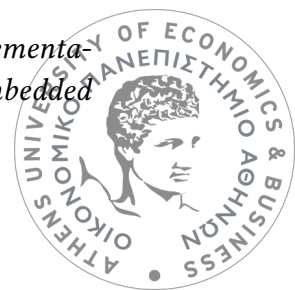
- [Sha19d] Tushar Sharma. Designite validation data, January 2019.
- [Sha19e] Tushar Sharma. Puppeteer, February 2019. <https://github.com/tushar-tushar/Puppeteer>.
- [SHK⁺14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.
- [SJW08] Lok Fang Fang Stella, Stan Jarzabek, and Bimlesh Wadhwa. A comparative study of maintainability of web applications on J2EE, .NET and ruby on rails. In *Proceedings - 10th IEEE International Symposium on Web Site Evolution, WSE 2008*, pages 93–99. National University of Singapore, Singapore City, Singapore, IEEE, December 2008.
- [SK17] Satwinder Singh and Sharanpreet Kaur. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*, pages –, 2017.
- [SKBD14] Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. Code-Smell Detection as a Bilevel Problem. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(1):6–44, October 2014.
- [SKS⁺15] Tara N Sainath, Brian Kingsbury, George Saon, Hagen Soltau, Abdel-rahman Mohamed, George Dahl, and Bhuvana Ramabhadran. Deep convolutional neural networks for large-scale speech tasks. *Neural Networks*, 64:39–48, 2015.
- [SLJ⁺15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
- [SLT06] Mazeiar Salehie, Shimin Li, and Ladan Tahvildari. A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 159–168. University of Waterloo, IEEE Computer Society, June 2006.
- [SM06] Scott Stribrny and Fran Boehme Mackin. When politics overshadow software quality. *IEEE Software*, 23(5):72–73, September 2006.
- [Smi00] C Smith. Software performance antipatterns. In *Proceedings Second International Workshop on Software and Performance WOSP 2000*, pages 127–136a. Performance Engineering Services, Santa Fe, United States, December 2000.



- [SMT16] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite — A Software Design Quality Assessment Tool. In *Proceedings of the First International Workshop on Bringing Architecture Design Thinking into Developers' Daily Activities*, BRIDGE '16. ACM, 2016.
- [Son16] SonarQube. <http://www.sonarqube.org/>, 2016. [Online; accessed 25-Oct-2016].
- [Spi15] Diomidis Spinellis. Tools and Techniques for Analyzing Product and Process Data. In Tim Menzies, Christian Bird, and Thomas Zimmermann, editors, *The Art and Science of Analyzing Software Data*, pages 161–212. Morgan-Kaufmann, 2015.
- [Spi19] Diomidis Spinellis. dspinellis/tokenizer: Version 1.1, February 2019. <https://github.com/dspinellis/tokenizer>.
- [SS17] Tushar Sharma and Diomidis Spinellis. Selected Resources for a Literature Survey on Software Smells, November 2017.
- [SS18] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158 – 173, 2018.
- [SSN12] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.
- [SSS14] Girish Suryanarayana, Ganesh Samarthayam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, 1 edition, 2014.
- [SSS15] Girish Suryanarayana, Tushar Sharma, and Ganesh Samarthayam. Software Process versus Design Quality: Tug of War? *IEEE Software*, 32(4):7–11, 2015.
- [SSS16] Ganesh Samarthayam, Girish Suryanarayana, and Tushar Sharma. Refactoring for software architecture smells. In *Proceedings of the 1st International Workshop on Software Refactoring*, pages 1–4. ACM, 2016.
- [SSSG13] Ganesh Samarthayam, Girish Suryanarayana, Tushar Sharma, and Shrinath Gupta. Midas: A design quality assessment method for industrial software. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 911–920, 2013.
- [Sty16] Puppet language style guide. https://docs.puppetlabs.com/guides/style_guide.html, 2016. [Online; accessed 22-Jan-2016].
- [SVT16] Marcelino Campos Oliveira Silva, Marco Tulio Valente, and Ricardo Terra. Does technical debt lead to the rejection of pull requests? [abs/1604.01450](https://arxiv.org/abs/1604.01450), 2016.



- [SYA⁺13] Dag IK Sjöberg, Aiko Yamashita, Bente Anda, Audris Mockus, and Tore Dyba. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, August 2013.
- [SYKG16] Zephyrin Soh, Aiko Yamashita, Foutse Khomh, and Yann-Gaël Guéhéneuc. Do Code Smells Impact the Effort of Different Maintenance Programming Activities? In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 393–402. IEEE, 2016.
- [SZV⁺13] Rodrigo O. Spínola, Nico Zazworka, Antonio Vetrò, Carolyn Seaman, and Forrest Shull. Investigating technical debt folklore: Shedding some light on technical debt opinion. In *Proceedings of the 4th International Workshop on Managing Technical Debt, MTD '13*, pages 1–7. IEEE Press, 2013.
- [TAV13] Edith Tom, Aybüke Aurum, and Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498 – 1516, 2013.
- [TC11] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems & Software*, 84(10):1757–1782, October 2011.
- [TCC08] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331. University of Macedonia, IEEE Computer Society, April 2008.
- [TGPM17] Alexandre Torres, Renata Galante, Marcelo Soares Pimenta, and Alexandre Jonatan B. Martins. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information & Software Technology*, 82:1–18, 2017.
- [TK11] Catia Trubiani and Anne Koziolk. Detection and solution of software performance antipatterns in palladio architectural models. In *ICPE '11: Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, pages 11–11. Karlsruhe Institute of Technology, ACM, March 2011.
- [TME⁺18] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, pages 483–494. ACM, 2018.
- [VAPM13] A. Vetrò, L. Ardito, G. Procaccianti, and M. Morisio. *Definition, implementation and validation of energy code smells: an exploratory study on an embedded system*, pages 34–39. ThinkMind, 2013.



- [VCD17] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 683–693. ACM, 2017.
- [VEM02] E Van Emden and L Moonen. Java quality assurance by detecting code smells. *Ninth Working Conference on Reverse Engineering*, pages 97–106, 2002.
- [vEM12] E. van Emden and L. Moonen. Assuring software quality by code smell detection. In *2012 19th Working Conference on Reverse Engineering*, Oct 2012.
- [VMDP14] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23(3):501–532, 2014.
- [VRDBDR07] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. *IEEE Transactions on Software Engineering*, 33(12):800–817, December 2007.
- [VVDP⁺16] Santiago Vidal, Hernan Vazquez, J Andrés Díaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. JSpIRIT: A flexible tool for the analysis of code smells. In *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, pages 1–6. Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina, IEEE, February 2016.
- [Wak03] William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., 1 edition, 2003.
- [WFF18] Bartosz Walter, Francesca Arcelli Fontana, and Vincenzo Ferme. Code smells and their collocations: A large-scale experiment on open-source systems. *Journal of Systems and Software*, 144:1 – 21, 2018.
- [WGM⁺15] Tsung-Hsien Wen, Milica Gasic, Nikola Mrkšić, Pei-Hao Su, David Vandyke, and Steve Young. Semantically conditioned lstm-based natural language generation for spoken dialogue systems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1711–1721, 2015.
- [WHTK14] Chunyan Wang, Shoichi Hirasawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. A Platform-Specific Code Smell Alert System for High Performance Computing Applications. In *IPDPSW '14: Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 652–661. IEEE Computer Society, May 2014.
- [WHZ⁺16] Yequan Wang, Minlie Huang, Li Zhao, et al. Attention-based lstm for aspect-level sentiment classification. In *Proceedings of the 2016 conference on empirical methods in natural language processing*, pages 606–615, 2016.



- [WL17] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pages 3034–3040, 2017.
- [WTVP16] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.
- [WVLVP15] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE Press, 2015.
- [Yam14] Aiko Yamashita. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. *Empirical Software Engineering*, 19(4):1111–1143, 2014.
- [YC13] Aiko Yamashita and Steve Counsell. Code smells as system-level indicators of maintainability: An empirical study. *The Journal of System and Software*, 86(10):2639–2653, October 2013.
- [YM13a] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 682–691. IEEE Press, 2013.
- [YM13b] Aiko Yamashita and Leon Moonen. To what extent can maintenance problems be predicted by code smell detection? – An empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013.
- [YN17] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 440–450, 2017.
- [YZFW15] Aiko Yamashita, Marco Zanoni, Francesca Arcelli Fontana, and Bartosz Walter. Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, pages 121–130. Høgskolen i Oslo og Akershus, Oslo, Norway, IEEE, November 2015.
- [ZHB11] Min Zhang, Tracy Hall, and Nathan Baddoo. Code Bad Smells: A review of current knowledge. *Journal of Software Maintenance and Evolution*, 23(3):179–202, April 2011.



- [ZSSS11] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. Investigating the impact of design debt on software quality. In *MTD '11: Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 17–23. Fraunhofer USA, Inc., ACM, May 2011.



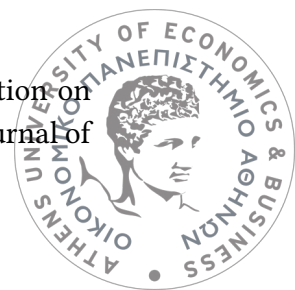
List of Publications

Accepted publications based on this thesis

- Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). ACM, New York, NY, USA, 189-200. <https://doi.org/10.1145/2901739.2901761>
- Tushar Sharma, Diomidis Spinellis. "A survey on software smells", Journal of Systems and Software, Volume 138, 2018, Pages 158-173, ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2017.12.034>.
- Tushar Sharma, Marios Fragkoulis and Diomidis Spinellis, "House of Cards: Code Smells in Open-Source C# Repositories," 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, 2017, pp. 424-429. [doi:10.1109/ESEM.2017.57](https://doi.org/10.1109/ESEM.2017.57)
- Tushar Sharma, Marios Fragkoulis, Stamatia Rizou, Magiel Bruntink, and Diomidis Spinellis. "Smelly relations: measuring and understanding database schema quality", In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18). ACM, New York, NY, USA, 55-64. <https://doi.org/10.1145/3183519.3183529>
- Tushar Sharma. 2018. Detecting and managing code smells: research and practice. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE '18). ACM, New York, NY, USA, 546-547. <https://doi.org/10.1145/3183440.3183460>
- Tushar Sharma. How Deep Is the Mud: Fathoming Architecture Technical Debt Using Designite. To appear in International Conference of Technical Debt (TechDebt'19), Tools track.

Submitted articles based on this thesis

- Tushar Sharma, Paramvir Singh, Diomidis Spinellis, "An Empirical Investigation on the Relationship between Design and Architecture Smells" under review in Journal of



Software and Systems (JSS). Apr 2018.

- Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. On the Feasibility of Transfer-learning Code Smells using Deep Learning. April 2019. Eprint available at: <https://arxiv.org/abs/1904.03031>SSL

Relevant accepted publications not part of the thesis

- Tushar Sharma, Pratibha Mishra and Rohit Tiwari, “Designite - A Software Design Quality Assessment Tool,” IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers’ Daily Activities (BRIDGE), Austin, TX, 2016, pp. 1-4. doi: 10.1109/Bridge.2016.009
- Tushar Sharma, “Designite: A Customizable Tool for Smell Mining in C# Repositories”, in SATToSE, Madrid, 7-9 June 2017. Online: http://www.tusharma.in/preprints/designite_SATToSE2017.pdf
- Ganesh Samarthayam, Girish Suryanarayana, and Tushar Sharma. “Refactoring for software architecture smells”, In Proceedings of the 1st International Workshop on Software Refactoring (IWoR 2016). ACM, New York, NY, USA, 1-4. <http://dx.doi.org/10.1145/2975945.2975946>

Other publications during the thesis period

- Tushar Sharma and Girish Suryanarayana, “Augur: Incorporating Hidden Dependencies and Variable Granularity in Change Impact Analysis,” 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), Raleigh, NC, 2016, pp. 73-78. doi: 10.1109/SCAM.2016.32
- Maria Kechagia, Tushar Sharma and Diomidis Spinellis, “Towards a Context Dependent Java Exceptions Hierarchy,” 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, 2017, pp. 347-349. doi: 10.1109/ICSE-C.2017.134

