Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Dipartimento di Elettronica, Informazione e Bioingegneria

POLITECNICO
MILANO 1863

Detecting Code and Community Smells in Open-Source: an automated approach

Relatore:	Prof.ssa Elisabetta Di Nitto
Correlatore:	Dr. Damian Andrew Tamburri

Tesi di laurea di:

Francesco Giarola Matr. 820446

Anno Accademico 2016–2017

To G.
my mentor.

Acknowledgements

Above all, I would like to thank professors Elisabetta Di Nitto and Damian Andrew Tamburri, for their support and priceless advice in the creation, development and realisation of this master thesis. I would also like to say thank you to my close and distant friends who patiently devoted their time and effort to me.

Finally, I would like to thank everyone who helped me and made all of this possible.

Francesco

Abstract

In this post-financial crisis era the software industry has became more competitive than ever, in such a scenario quality and speed are probably the two most important unit of measure of both success and healthiness of a software community and their products.

Software development and software engineering are now more than ever a community effort because their success often depends on people and their socio-technical characteristics.

Therefore, it becomes fundamental being able to measure and balance consequently the level of quality at the pace that the global market requires.

To offer support in these situations from both Technical and Social point of view, we conducted an empirical research in order to define, operationalise and evaluate the correlation between Code and Community Smells with the aim of providing to community leaders the most useful insight when trying to access the overall level of quality.

The proposed solution provides the identification and correlation between Code and Community Smells, constituting a tool that can be used for continuous Social and Technical Debt management and improvement.

We evaluated our tool on ten Open Source development communities and made some key findings concerning the correlation between organisational and technical quality factors mainly related to the lack of communication between community members and its effects on the quality of the produced code.

Sommario

In questa era post crisi economica il settore dello sviluppo software è diventato più competitivo che mai, in questo scenario qualità e velocità sono probabilmente le due più importanti unità di misura di successo e salute di una comunità software e dei prodotti che sviluppa.

L'ingegneria e lo sviluppo del software sono ora più che mai uno sforzo comunitario, dal momento che il loro successo dipende dalle persone e dalle loro caratteristiche socio-tecniche.

E' dunque diventato fondamentale poter misurare e bilanciare di conseguenza il livello qualitativo al ritmo che il mercato globale richiede.

Per offrire un supporto in queste situazioni dal punto di vista sia Tecnico che Sociale, è stata condotta una ricerca empirica al fine di definire, operazionalizzare e valutare la correlazione tra Code e Community Smells con l'intento di fornire a chi guida una comunità e tenta di misurarne il livello di qualità quanti più consigli utili possibile.

La soluzione proposta fornisce l'identificazione e la correlazione tra Code e Community Smells, costituendo uno strumento che può essere utilizzato per migliorare e gestire in modo continuo il Social e il Technical Debt.

Lo strumento è stato valutato in dieci comunità di sviluppo Open Source e sono stati individuati alcuni fattori qualitativi relativi alla correlazione tra aspetti organizzativi e tecnici prevalentemente legati alla mancanza di comunicazione tra i membri della comunità e ai suoi effetti sulla qualità del codice prodotto.

Contents

Chapter 1

Introduction

In the last decade, software became predominantly engineered by large and globally-distributed communities and consequently, now more than ever, it is of vital importance knowing more on the quality of these communities to ensure the success of a software project [55]. Socio-technical decisions, like changing the organisational structure of a software development community or its internal development processes (e.g., adopting agile methods), modify how people work and interact with each other and, as a side effect, they influence the well-being and success of the software project [69].

Previous researches revealed that software development communities can develop ills that collectively contribute to a form of additional project cost that was defined Social Debt [68], which is similar but parallel to Technical Debt [45], because it represents additional project costs not necessarily related to the source code itself but rather to its "social" nature and thus it is correlated to sub-optimal organisational structure and socio-technical characteristics of a software community.

Unforeseen project costs connected to sub-optimal development decisions or community habits, previously introduced as Technical Debt and Social Debt, on the long run can lead to the raising of both development and organisational anti-patterns. Such negative anti-patterns are know in literature as Code Smell [36] and Community Smell [70].

Since 2008 global economical crisis the world is changing at an incredibly fast pace, Information Technology industry is following that trend with the new concept of DevOps. The DevOps movement was born with the purpose of shortening the time in between the development of a software product and the moment in which it became operative. As for any another software artefacts DevOps output is and want to be measured in order to increase the overall quality [18]. Therefore automated tools has been introduced to measure both Technical and Social [50] quality level, the problem we want to solve in the context of this master thesis is to connect both Technical and Social measures in an automated way so that we can have a wider and deeper perspective of the level of quality of both a software product and the community who develop it.

This master thesis elaborates, operationalises, validates and discusses the existence of correlations between Technical and Social Debts in the context of open-source software development communities. To the best of our knowledge, this kind of automated process is the first of its kind and may well inspire further research in the intriguing social software engineering field of managing Social Debt in addition to the canonical Tech Debt management.

We proceeded by formulating several hypotheses on potential correlations between specific Social Debt and the occurrence of Technical Debt defined within the model and then we evaluated our hypotheses against our corpus of data, consisting of occurrences of both Code and Community Smells for ten analysed Open Source Software development communities.

As a result of our evaluation, we found some quite logic but still valuable insights to assess the quality of software development communities. For example, considering the literature [57] we found out that Community Smells related to communication lacks lead to the raise of Code Smells that are typical of lone developers works.

Chapter 2 discusses the state of the art of important aspects that were fundamental in the definition and elaboration of this master thesis. Chapter 3 provides an overview of the problem analysis and research questions that are at the foundation of this work. Our implementation of Community Smells' identification patterns and quality factors belonging to our Socio-technical Quality Framework is proposed and explained in Chapter 4. Further on, Chapter 5 provides the evaluation of our work with the relative findings related to the correlation between Code and Community Smells within FLOSS development communities and Chapter 6 concludes this master thesis.

Chapter 2

State of the art

This chapter introduces background information and related work which were fundamental in the formulation and execution of this master thesis. The state of the art of Conway's law research field, presented in Section 2.1, provided the theoretical concepts and hypothesis at the foundation of this research. Sections related to Global Software Development and to its most particular case constituted by Free/Libre and Open Source Software, discussed respectively in Section 2.2 and Section 2.3, are introduced to provide the necessary background information to understand the context of our empirical research; furthermore, the provided concepts were fundamental to identify potential socio-technical issues that are intrinsic within the two typologies of development environments studied within the literature, in order to further comprise and define quality factors capable of capturing every aspect of a software development community and their associated side effects within our framework. Section 2.4 presents a set of researches, in order to demonstrate the validity, effectiveness and efficacy of using Developer Social Networks in empirical software engineering researches, build considering either mailing lists and Version Control Systems. Social Debt and its technical counterpart are introduced in Section 2.5, to ensure a deeper understanding of the main topics covered within this master thesis and to identify potential quality factors capable of impacting the health of a software development community (e.g., communicability). Section 2.6 summarises two important software engineering researches that provided both the theoretical and practical foundations and verified the effectiveness and validity of the applied empirical research approach of this master thesis.

2.1 Conway's law and beyond

In 1968 with his article titled "How do committees invent?" [29], Dr. Melvin Conway introduced for the first time the idea, now commonly called "*Conway's law*", that systems designed by an organisation are constrained to produce designs which are copies of the communication structure of the same organisation. Conway, through the use of linear-graph notation, demonstrated that there is a very close relationship between the structure of a system and the structure of the organisation which designed it. The consequence of this homomorphism is that if subsystems do have their own separate design group then the structure of each design group and the system's organisation will be identical, otherwise if the same group designed multiple subsystems, every subsystem's structure will have the same design group collapsed into one node representing that group. The phenomenon described by Conway's law is more evident as the organisation size increases, because its flexibility diminishes.

Software development is characterised by a technical and a social component. The technical component is composed by the processes, tasks and technologies used during the software development, while the social component is constituted by organisations and people involved in the development and their characteristics. Due to this dichotomy, *software development can be considered a social-technical activity*, in which the technical and the social components need to be aligned in order to succeed [24].

To design a computer program or any other type of artefact, the initial steps are more related to design activity rather then to the system itself since the design activity cannot proceed until its boundaries and the boundaries of the system to be defined are understood and until a preliminary notion of the system's organisation is achieved. As a consequence of this Conway concluded that "the very act of organising a design team means that certain design decisions have already been made, explicitly or otherwise". The steps after the choice of such preliminary system concepts are [29]: organisation of the design activity and delegation of tasks according to that concept, coordination among delegated tasks and consolidation of sub-designs into a single design. A system is then structured from the interconnection of smaller subsystems, and so on, until a stage in which the subsystems are easy enough to be understood without further subdivisions is reached. Large systems naturally tend to disintegrate themselves more than small systems during the development activities and so a system management activity should be used to mitigate this dangerous characteristic. To achieve an effective coordination among teams, architecture is not the only dimension that should be considered but even plans, processes and coordination mechanisms are fundamental elements [39].

Fred Brooks in his book titled "The Mythical Man-Month" [22], in agreement with Melvin Conway's theory, verified that the product quality is strongly related to the organisational structure.

Due to the homomorphic relation between components and the organisational structure, Conway [29] proposed the theory that a team can work on many components but that a single module must be assigned to a single team. Different modules can be developed in parallel and independently from each other and the development time should be shortened since separate teams work on different modules and, as a consequence, the communication need is reduced. Wong et al. [75] went further in this direction and created a model for reasoning and making prediction about design structure and the consequences in coordination requirements.

Since the beginning of software development, metrics were defined to estimate the quality of developed software (e.g. LOC, code churn, code complexity, code dependencies) but they measured only the technical aspect of software and ignored the "social" factor of software development which is related to people and to the organisational structure. Using Brooks' theory as a starting point, Nagappan et al. [55] analysed the relation between organisational structure and software quality. They proposed eight measures to quantify organisational complexity from the code viewpoint and empirically evaluated their efficacy to identify failure-prone binaries in a commercial project. The failure-proneness prediction model based on the organisational metrics outperformed traditional technical metrics (e.g. code churn, code complexity, LOC).

The concept of socio-technical congruence was introduced by Cataldo et al. as the "match between the coordination requirements established by the dependencies among tasks and the actual coordination activities carried out by the engineers" [24, 25]. Cataldo et al. discovered that *socio-technical congruence is highly correlated to the software development productivity*: a higher socio-technical congruence is proven to speed-up software development, reducing the amount of time needed to perform a task and they demonstrated that over time developers learn to use the available communication channels in such a way to reach a higher congruence, thus if the development base is stable then the socio-technical congruence should increase over time [25].

The concept of socio-technical congruence was later redefined in 2008 by Sarma et al. as "the state in which a software development organisation harbours sufficient coordination capabilities to meet the coordination demands of the technical products under development" [64] and the following socio-technical congruence characteristics were identified:

1. it represents a *state* because it captures a particular moment in time of the company's social and technical context;
2. it is *descriptive* of a certain state in which an organisation finds itself in a defined time because individuals that perform the work may take non-optimal decisions;
3. it is *dynamic* because the technical and social structures change and evolve over time;
4. it is *multi-dimensional* because it depends on every possible way to coordinate

work;

5. it can be considered at *multiple levels*: individuals, sub-teams, teams or entire organisation;

6. it i*nvolves trade-offs* because congruence may differ in every considered level and achieving congruence in a level may create an incongruence in another one.

In 2010 Colfer and Baldwin [28] complemented Conway's law verifying the validity of their *mirroring hypothesis*, which assumed that the organisational patters of a development community (e.g. team co-membership and geographic distribution, communication links) mirror the technical dependency patterns of the software under development. Their contribution added the opposite causality relationship to Conway's law: *the technical structure mirrors the organisational structure*, essentially turning Conway's original argument into an *isomorphism.* In other words, a change to the communication structure will eventually trigger a change to the design structure to return the socio-technical system into a state of socio-technical congruence.

The relation between software development organisational changes (e.g. forks, company acquisition, open-sourcing) and software quality was addressed even by Sato et al. [65], who demonstrated that when multiple organisations (concurrently or in temporal succession) modify the same file, the increased modification frequency and complexity will lead the file to be more faulty.

Summing up, socio-technical congruence states that if two people work on dependent tasks then they need to communicate with each other. Communication needs can be computed analysing code modules and their inter-dependences. For example, if two developers work on inter-dependent modules then they have to communicate to coordinate their work and if they do not communicate and this gap is detected, then it can suggests a coordination problem. Socio-technical congruence management consists in reducing the number of this kind of gaps. To minimise those gaps it is possible to promote coordination mechanisms or reducing the coordination needs (e.g. reducing modules inter-dependences [61]).

2.2 Global Software Development

One of the main innovations that is characterising the 21th century is globalisation: "the process of international integration arising from the interchange of world views, products, ideas and mutual sharing, and other aspects of culture" [1]. Globalisation is influencing every aspect of our life, from politic to economy, from society to technological systems, through the connection and integration of companies, people and nations on a global scale.

Friedman in his book “The World Is Flat: A brief history of the twenty-first century” defined ten “flattener” events that allowed all commercial competitors to have the same opportunities by playing with the same set of rules, enabling the globalisation process. Those historical events are [37]:

1. *11 September 1989 – “Fall of Berlin wall”*: represents the end of the Cold War and the revolutionary possibility of creating personal software programs, contents and interconnections with other people around the world using Personal Computers;

2. *8 September 1995 – “Netscape”*: Internet is accessible to everyone;

3. *“Work-flow software”*: virtual applications are now able to cooperate without human assistance. This is considered by Friedman the “genesis” because it is the moment in time where the global platform enabling multiple forms of collaboration was born;

4. *“Open-sourcing”*: communities collaborate and upload their work on on-line projects;

5. *“Outsourcing”*: companies can split and externalise their activities in efficient and effective ways;

6. *“Off-shoring”*: international relocation of company’s processes in countries where production costs are lower;

7. *“Supply-chaining”*: supply and demand management is integrated across companies;

8. *“In-sourcing”*: Commercial companies employees perform services for connected third party companies;

9. *“Informing”*: social and search engines and information-rich websites allow access to a massive amount of information;

10. *“The steroids”*: any analogical content can be digitised and telematically transmitted at high speed, in mobility, any time and by anyone.

As a consequence of outsourcing and off-shoring flatteners, commercial software started to be developed by different geographically distributed and cooperative commercial software companies. There two flatteners are the fundamental prerequisites to enable Global Software Development (GSD), that is defined as “the nature of globalisation which reduces temporal, geographic, social, and cultural distance across countries” [26].

Global Software Development differs from traditional software development because in addition to the customer that buys the software and the commercial company that sells it, there are suppliers whom develop software through the mechanism of outsourcing and off-shoring. Global Software Development can be attractive for commercial companies because it reduces production costs due to its off-shoring nature, it allows companies to hire the best developers from any country of the world, it creates the chance of constitute virtual corporations in very fast ways, it allows to benefit from proximity to the market and it enables a "round the clock" software development approach, through the exploitation of different time zones, improving the time-to-market.

In a Global Software Development environment, as previously seen, the lack of communications between developers and the assignment of the same task to two different geographical sites can compromise the development success. Considering the off-shoring characteristic of GSD, social and cultural differences between developers can impact on the overall trust and software quality. To achieve the best performance from a GSD approach, commercial companies should take some precautions to avoid potential side effects, that can be categorised largely as temporal, geographical, social and cultural barriers [26]. Some useful strategies to limit Global Software Development side effects are: communication and coordination executed through common processes, strategic sub-division of tasks [61], offer cultural education to employees, understand diversity and taking advantage from it [73].

Diversity arises from attributes that differentiate people as their demographic information (e.g., gender, nationality, age), their functional information (e.g., role, knowledge, expertise) or their subjective information (e.g., personality, ethic). Molleman et al. [54] considered team diversity by addressing demographic characteristics, personality traits, technical skills and knowledge characteristics and analysed their impact on team functioning and performance in industrial manufacturing and service environments. *The characteristics of a team can be considered at three different levels: global, shared and compositional.* Global characteristics can be measured at team level (e.g., time size), shared characteristics are related to individual team members perceptions that tend to be shared by all the other team members (e.g. mutual trust) and compositional characteristics are related to individual team members attributes (e.g., age, skills). Within global team characteristics Molleman et al. considered team size and verified the intuitive idea that the optimal team size depends on the team tasks and discovered that "if workers are independent or only have to share resources such as tools, a larger team will achieve a better performance" because team tasks will be simpler and require less coordination effort. This result is similar to the one obtained by Parnas [61] and other researches reported in Section 2.1. Molleman et al. concluded that demographical similarity (e.g., gender, age) facilitates team functioning and effectiveness, enhancing mutual linking and trust.

On the opposite side demographic diversity can cause cliquishness, stereotyping and subgroups conflicts [54]. Earley et al. [34] discovered that even if team diversity negatively impacts team functioning and effectiveness in the short-medium term, its side-effect tends to be less relevant as time passes because a common identity will be created with the institution of ways to interact and communicate.

In conclusion, the increasing interest in Global Software Development created new generations of "software development processes, practices and trends such as ubiquitous computing, agile methodologies, project outsourcing, distributed software development, process improvement and standardisation, mobile applications development, social networking, and process tailoring practices" [73]. These new typologies of software applications generated new software development trends and styles that should be implemented by commercial companies to improve their efficiency and effectiveness in software development (e.g., agile methods).

2.3 Free/Libre and Open Source Software

In February 1986 Richard Stallman, founder of the Free Software Foundation (FSF), defined "*Free Software*" as any software that respects user and community freedom, allowing users to be free to run, copy, study, change, improve and distribute the software. Free software is an ethical matter of liberty and freedom, it is not related to price. Free Software does not mean non-commercial and a free software program must be available for commercial use, development and distribution. To highlight the fundamental idea that it does not mean gratis, sometimes Free Software is called Free/Libre Software, adding the French or Spanish word that means free in the sense of freedom. Four fundamental freedoms were specified to define Free Software with the purpose of allowing users to control the program and what it can do for them. The four freedoms to classify a software as Free Software are [2]:

1. The freedom to run the program as you wish, for any purpose (*Freedom 0*);
2. The freedom to study how the program works and possibility to change it so it will compute as you wish (*Freedom 1*). Access to the source code is a precondition for this freedom;
3. The freedom to re-distribute copies, so you can help your neighbours (*Freedom 2*);
4. The freedom to distribute copies of your modified versions to others (*Freedom 3*). By doing this you can give to the whole community a chance to benefit from your changes. Access to the source code is a precondition for this freedom.

Free Software Foundation's social activism and the misunderstanding of the word "Free" were considered not appealing to commercial software companies by some developers and to promote the potential business deriving from the collaboration and the sharing of source code, the term "*Open Source*" was created and in February 1998 the Open Source Initiative was founded. A computer software is classified as Open Source Software (OSS) if its source code is available and it is licensed to provide the rights to study, change and distribute the software for any purpose. The Open Source Initiative states that Open Source does not just mean granting access to the source code but the software must obey to the following ten criteria [3]:

1. *Free re-distribution*: the license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale;

2. *Source code*: the program must include source code and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with its source code, there must be a publicised means of obtaining the source code for no more than a reasonable reproduction cost, preferably downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a pre-processor or translator are not allowed;

3. *Derived works*: the license must allow modifications and derived works and must allow them to be distributed under the same terms as the license of the original software;

4. *Integrity of the author's source code*: the license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software;

5. *No discrimination against people or groups*: the license must not discriminate against any person or group of persons;

6. *No discrimination against fields of endeavour*: the license must not restrict anyone from making use of the program in a specific field of endeavour. For example, it may not restrict the program from being used in a business, or from being used for genetic research;

7. *Distribution of license*: the rights attached to the program must apply to all to whom the program is redistributed, without the need for execution of an additional license by those parties;

8. *License must not be specific to a product*: the rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution;

9. *License must not restrict other software*: the license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other software distributed on the same medium must be open-source software;

10. *License must be technology-neutral*: no provision of the license may be predicated on any individual technology or style of interface.

Even if legally Free Software is qualified as Open Source Software, the Free Software Foundation consider the term Open Source Software close but not identical to Free Software as the word "Open" never refers to freedom, which is one fundamental component of the Free Software definition [2]. Stallman said that the two terms describe almost the same category of software, but they stand for views based on fundamentally different values: *Free Software is a social movement and Open Source Software is a development methodology.* He classifies the Free Software as an ethical imperative to respect the user freedom while Open Source concern is how improve software and increasing its popularity and success [4]. As the main difference between these definitions can be defined as political, in situations where the developer political views are not considered important it is possible to be neutral using the term *Free/Libre and Open Source Software* (FLOSS). The term Free and Open Source Software (FOSS) can also be used but the Free Software Foundation considers it misleading because it fails to explain that "free" refers to freedom [5]. In this master thesis we will use all the above definitions as synonyms.

An Open Source Community can be defined as a network platform in which the source code of the software is opened under an Open Source Software license agreement. Open Source Communities are fundamental for Open Source Software's promotion and development. In the last decade the interest in FLOSS grew widely and now many commercial companies develop, maintain and distribute their products through an Open Source Community (e.g. MySQL, Firefox). The pervasive diffusion and growing interest in FLOSS projects from both voluntary developers and commercial software companies constitute a precious asset since developers can extend, modify or reuse code from already existing projects. This possibility can increase developers productivity and reduce development costs.

FLOSS development can be driven by voluntary developers encouraging knowledge sharing rather than the protection of intellectual property, commercial companies and institutions from every part of the world. FLOSS software development is a perfect example of Global Software Development and it has all the GSD pros and cons. Group dynamics in FLOSS communities are substantially different from commercial off-line teams, for example [72]:

- *Geographic dispersion* and cultural differences are the norm, as community members rarely meet in person;
- Collaborators assemble in *on-line communities* and coordinate their activities through distributed communication channels (e.g., mail lists);
- *Teams are fluid*: they tend to form and dissolve organically around a specific task;
- *High turnover* since FLOSS contributors are often volunteers;
- FLOSS communities are generally constituted by a set of *core developers* and a more loosely coupled group of contributors that support the development by reporting issues, submitting patches or contributing with documentation (core-periphery structure).

Open Source communities have an implicit diverse nature as they are usually composed by a variety of contributors ranging from volunteers to developers sponsored by companies and all of them have different demographic characteristics, knowledge, personalities, skills, cultures and educations. Open Source Software projects can benefit from their intrinsic diversity since it stimulates creativity, diversity of ideas and problem solving skills coming from different background and knowledge, therefore increasing global projects productivity [72]. On the other hand if diversity increases but it is not managed, it may create conflicts within the development team negatively effecting the team's cohesiveness and its performance, due to greater perceived differences in values, ideas, norms and communication style. [54].

Software licenses define under which conditions software can be used, copied, modified or redistributed without incurring in legal problems. As proprietary licensing tend to restrict the possible ways in which a software will be used, Open Source licenses tend to limit the restrictions that can be associated to a software, to ensure development freedom and source code re-distribution. Usually the license under which a project is released can be explicated in three different ways:

1. Adding a licensing comment on top of each file of the project. This approach allows a fine-grained license definition;
2. A specific file specifies the license under which the software is released;
3. The software license is expressed within the project's official website.

Several FLOSS licenses exist, from highly restrictive (e.g., GPL) to more flexible (e.g., MIT), but their main goal is to promote and enable the right to fork, copy, modify and redistribute the program source code. As it evolves a project can change its license to better meet the requirements and needs of the development community or of external actors interested in the project. Vendome et al. [74] highlighted that the initial software license is influenced by the communities to which core developers are already contributing and that external actors do not have impact in this choice. As projects grow their current licenses are heavily affected by their need to be commercially reused and, to accomplish this purpose, Open Source projects tend to migrate toward less restrictive and more permissive licenses. Licenses do not only define how software code source can be reused but they might also affect other components of the projects where the FLOSS code will be used. For example GPL license requires that all the source code, in which a GPL licensed component is used, has to be released under the GPL license. More flexible licenses (e.g., MIT license) do not require this condition and allow the use of FLOSS code under any other license, including commercial use. This license dependency was found even by Vendome et al. [74] who stated that a license change of a sub-component might start a chain reaction that will influence the final project's license or will cause the drop of the sub-component usage due to incompatibilities between licenses.

The number of commercial driven Open Source Projects is increasing year after year, *FLOSS development has long become an important commercial activity* and the Open Source Software ecosystem is full of successful projects which are completely driven by commercial companies (e.g., Android) or which have developers sponsored by commercial companies (e.g., Linux). When a voluntary-based Open Source Software project become promising, commercial companies may be interested in participating in its development, to adequate the software to their needs. It is possible to consider the ratio of volunteer to paid work as an indicator for the health of FLOSS projects and it can aid project leaders in managing their community [63]. Riehle et al. [63] in 2014 discovered that even if Open Source Software has been growing near-exponentially, its global ratio of voluntary and paid development is almost constant. A possible explanation is that for every project which increases its economic significance receiving sponsored development, a new totally voluntary driven project is started. Voluntary FLOSS developers, in contrast to commercial software developers, usually experiment a high degree of development and organisational freedom with respect to the possible ways through which contribute to the project and how to organise themselves and their tasks.

2.4 Developer Social Networks

In a software development community every interaction and relationship between developers can be modelled through a self-organised network, which can be considered as a latent developer Social Network [40]. In such developer Social Network, considering the case of FLOSS projects, developers and their relationships are subjects to continuous variations and changes as the set of active developers and their activities change over time.

A Developer Social Networks (DSN) can be modelled through the use of nodes, that represent actors, and edges, that represent relationships between different actors (or groups). A Social Network and its actors have two fundamental properties: connection and distance [47]. *Connectivity* can be measured using density, size, centrality and reachability of the Social Network. As a member of the Social Network is more connected then he or she is exposed to more information, can be considered more influential in the community and may be easily influenced by others. *Distance* in a Social Network represents the closeness of two actors within the network and may be a useful indicator to identify macro-properties differences, like diffusion and homogeneity. Distance influences the information diffusion time across the community and it can be measured using walks and paths. Connections and distances are fundamental characteristics to enable the identification of sub-communities within a Social Network, which are defined as "subsets of actors among whom there are relatively strong, direct, intense, frequent, or positive ties" [47].

Since Free/Libre Open Source Software communities number increase daily, the amount of open and accessible information about FLOSS development grow exponentially. Issue tracking systems [41], mailing lists [19] and code repository histories [49] of FLOSS projects can be easily and freely mined by researchers to analyse defects, communication and distributed collaboration habits of Open Source Software developers. The existence of FLOSS project is enabled by Internet that allows communication and coordination (C&C) activities between developers. Such activities are typically *public and accessible to anyone* and this allows researchers to track and mine coordination and communication activities and study them through the usage of Developer Social Networks, in contrast to industrial closed-source projects where C&C activities are predominantly direct and informal [19].

FLOSS projects are extremely interesting for empirical software engineering studies because they imply a distributed development occurring at a global scale and all the information related to a project (communications, code modifications, bugs, etc.) are available on-line (mailing lists, code repositories, bug tracking systems, etc.), granting the possibility of mining them. To take advantage of this enormous quantity of available data and to be able to mine and make sense of the organisational, social, technical and communicational aspects of a FLOSS project, researchers should refactor the retrieved information into a structured and analysable form. During last decade the main technique used to model technical and social aspects of software developers to enable the possibility of studying how people collaborate and organise their work in a global software development environment is the Social Network approach.

Public mailing lists are the classic channel used in FLOSS projects to perform communication and coordination activities and their archives (usually available online) in conjunction with VCS and other on-line development artefacts (e.g., bug tracking systems) allow researchers to create a developer Social Networks able to model and understand communication, coordination and collaboration practices and patterns in FLOSS projects.

During the last decade researchers have generated Developer Networks from every possible development source of information to enable the usage of Social Network Analysis methodologies and metrics. A Social Network can be created from code source history and mostly from any other kind of open and accessible data source used to support the software development (e.g., bug reporting [40], VCS [48] and mailing lists [19]). As an example, to conduct a knowledge-centric software engineering empirical study, the mailing list of a project should be considered: every member whom sent a message on the mailing list is considered a node and if a person A received a reply to one of his messages from another member B, then it exists an edge connecting A and B.

Conway law states that the project structure is strongly correlated to the organisational structure of the project, thus understanding the Developer Network is fundamental to estimate the quality and efficiency of software development activities. Social Network Analysis is based on individuals and how these individuals are related between them through relationships. In software engineering these relationships can be extracted mining software development artefacts, allowing to study all the possible ways in which people interact through all the available channels used to develop software. Empirical software engineering studies often apply SNA methodologies because they offer a solid systematic and quantitative framework.

FLOSS projects, due their intrinsic nature of being developed mainly by voluntary developers without a monetary retribution, can be affected by a sentiment of mistrust from companies that use them in their commercial activity. To avoid this phenomenon, in 2002, Madey et al. used a Social Networks approach to model FLOSS communities because "a better understanding of how the OSS community functions may help IT planners make more informed decisions and develop more effective strategies for using OSS software" [49].

One of the first attempt to use Social Network Analysis to analyse on-line communities was conducted in 2004 by Lin and Chen [47]. In their research study social ties, information flows, information and resource acquisition and coalitions creation were considered with the scope of accessing team collaborations, evaluate the performance of the system and enable the identification of relationships and interaction patterns within the community.

A socio-technical Developer Network can be created from socio-technical connections found exploiting the collaboration and communication channels and it can be analysed using SNA metrics. Social Network Analysis metrics calculated on socio-technical Developer Networks, created from connections observed in development artefacts, were proved to be representative of actual and real socio-technical relationships present within the software development communities [53]. Nia et al. [58] demonstrated that the effect of paths with broken information flow (consecutive edges which are out of temporal order) on the centrality measure of nodes within the network and the effect of missing links on such measures do not invalidate the Social Network Analysis metrics validity, but such metrics are stable with respect to such phenomenons. Betweenness centrality and clustering coefficient are stable in presence of a large number of missing links and this this essentially means that most of the activity in Developer Social Networks arise from few participants, thus it is sufficient to look at the 10% of developers [58].

In software engineering with the term *Version Control* (VC), it is considered any practice devoted to track and control changes to any possible project element: source code, documentation or configuration files. Since FLOSS development is intrinsically distributed and anybody can contribute modifying the source code, Version Control Systems (VCS) are extremely useful as they can track ownership of changes to the project source code. There are two main VCS typologies: centralised and distributed. *Centralised VCS* have a single central authoritative repository on which developers can synchronise their code-base; file locking and version merging are used to enable different developers to operate on the same file at the same time. Some famous Centralised VCS are: Concurrent Versions System (CVS) and Subversion (SVN). Opposed to the client-server approach of Centralised VCS, *Distributed VCS* implement a peer-to-peer approach as they don't have any central authoritative repository but the source code can be checkout and committed into any existing repository with a merge operation. Some famous Distributed VCS are: Git, Mercurial and Bazaar. Brindescu et al. [21] conducted an empirical software engineering study to compare the impact of Centralised VCS and Distributed VCS on software changes. They discovered that Distributed VCS have a smaller commit size in terms of lines of code and that hybrid repositories (repositories that migrated from a centralised to a distributed VCS) do not show any difference between the size of commits performed before and after the switch of paradigm due to commit policies formed in the team while using the centralised approach. In the past decade Distributed VCS saw an increase in popularity with respect to Centralised VCS and many popular FLOSS projects migrated from a centralised to a Distributed VCS. Distributed VCS have the following main differences respect to the centralised approach:

1. Only working copies exist because a reference copy of the code does not exist by default;
2. Every working copy is a remote backup of the change-history of the entire project;
3. It is possible to work without the need of being connected to a network;
4. Version Control operations are fast because no communication is needed;
5. Communications are necessary only when sharing a change between peers;
6. A web-of-trust approach can be used to merge changes coming from different repositories; this enables new work-flows that are impossible in Centralised VCS (e.g., intermediate roles can be responsible for integrating new changes proposed by developers);
7. Allow non-core developers (the ones who do not have write permissions on the repository) to contribute to the source code;
8. Authorship of changes of non-core developers is kept in historical records;
9. Individual changed lines of a file can be committed instead sending the whole file again;
10. Initial repository cloning is slower than Centralised check-out since all branches and change history are copied.

Version Control Systems (VCS) have been used to construct developer collaboration networks since the introduction of Social Networks and Social Network Analysis in empirical software engineering studies, due to their intrinsic capability of capturing inter-relationships among large software project components. Different VCS and VCS typologies will provide different grain level information to construct the collaboration network. For example Centralised VCS will provide only information about the committer, instead Distributed VCS will usually provide even information about the author of the commit. Since the information volume within a VCS can reach an incredible dimension, this can be considered a Big Data research area. Data retrieved from VCS is unusable without techniques to extract coherent information from this amount of data and highlight relevant trends and interesting aspects of a software project.

The first empirical software engineering research that considered VCS to generate a collaboration network was conducted by Lopez-Fernandez et al. in 2004 [48] and it proposed a set of Social Network Analysis methodologies to characterise the evolution and internal structure of FLOSS projects. They proposed to consider VCS committers or VCS directories (considered software modules) as nodes and the common commits as weighted edges between two nodes. In 2011 Jermakovics et al. [42] improved the methodology proposed by Lopez-Fernandez and allowed the generation of a more detailed and cleaner collaboration network, considering a file level grain instead of directories level to detect common commits (software modules). Developer networks generated using the methodologies proposed by Lopez-Fernandez or Jermakovics can be too dense and inefficient to obtain useful results during developer collaboration analysis. In 2015 Joblin et al. [44] addressed this problem and introduced a *collaboration network generation methodology that consider the code structure and detect when developers collaborated on the same function of a file, enabling a function level grain collaboration analysis.*

After 2004 every other possible development artefact was considered as a data source and used by researchers to create Developer Networks. Bird et al. [19] in 2006 were the first to exploit mailing list archives to construct a Developer Social Network of community members participating in a project. Always in 2006 Howison et al. [41] created for the first time in software engineering history a Developer Social Network from a bug reporting system. Both mailing lists and bug-tracking systems of FLOSS projects enable to explore communication and coordination activities of all the participants of a community and do not limiting the analysis just to software developers, as in the case of software code source (VCS) analysis, because mailing lists and bug-tracking systems contain many social interactions and bug reporting activities performed by users and people not necessarily directly involved in the software development (e.g., report bugs but do not provide patches).

Hong at al. [40] considered how and to what extreme Developer Social Networks can be analysed using General Social Networks (GSN) techniques, studied the evolution of Developer Social Networks in time and how the DSN topological structures can be influenced by project events (e.g., release, turnover). General Social Networks (e.g., Facebook, Twitter) as Developer Social Networks are founded on the freedom of participation but GSN offer more freedom of topics, while Developer Social Networks are mainly focused only on project development activities. Developer Social Networks are latent and not instantly usable, so they have to be extracted and constructed from information rich artefacts that support software development (e.g. VCS, mailing lists). Some other interesting aspects which characterise Developer Social Networks are [40]:

- DSN are usually characterised by a small portion of developers with high degree (core developers) and many developers with low degree, thus Developer Social Networks can be considered as *scale-free networks*;
- In DSN most pairs of developers can communicate or are connected between

each other through an exiguous number of hops in the network ("*small world*");

- DSN are *highly modular*, thus they do have a significant community structure, and modularity tend to increase over time.

Social Network Analysis methodologies and metrics were used in many empirical software engineering studies to implement models capable of predicting faults [55], failures [20], and vulnerabilities [66]. Nan and Kumar [56] took advantage of Social Networks Analysis to examine the joint effect of developer team structure and software architecture in Open Source Software and discovered that they moderate each other's effect on software development performance. Valetto et al. [71] applied Social Networks theories to Developer Social Networks and defined a useful methodology to compute socio-technical congruence, which is based on the direct comparison of the structure of an organisation with the project code source. In 2014 Jorge Colazo used Social Networks Analysis to analyse how collaboration DSNs change when collaborating teams become temporally dispersed and he discovered that "the collaboration structure networks of more temporally dispersed teams are sparser and more centralised, and these associations are stronger in those teams exhibiting higher relative performance" [27].

2.5 Technical and Social Debt

Since socio-technical decisions influence both the technical and social aspect of the software development environment, non-optimal or uninformed socio-technical decisions may generate additional costs to the technical or social area, or even both. Due to the nature of these additional costs, they can be considered as a debit because their resolution can be postponed in time since usually these non-optimal decisions are not easily detectable and visible.

Technical debt (TD) is a software engineering metaphor defined in 1992 by Cunningham [31] to describe the internal tasks that some decisions imply but that are not performed. If these tasks are not completed, the debt is not repaid and it will continue to accumulate interests, creating future problems and making it harder to implement changes in the future. A classical example of technical debt generated by a development team is when a decision that simplify a short term goal is taken but it has a great potential to negatively impact the development activity on the long term.

When a change in the source-code of a project is performed, it is often necessary to execute some other coordinated changes to other software components (e.g., other code modules, documentation) due to their inter-dependencies or due to development policies. Whenever this situation happen but the change associated to the software modification is delayed, technical debt arises and it must be paid off sooner or later in the future to avoid the failure of the software development.

Kruchten et al. redefined Technical Debt as “the invisible result of past decisions about software that negatively affect its future” [45], not limiting the concept to situations that imply a cost. Technical Debt is generated by invisible aspects of software aging and its evolution or it can be caused by external events. Some technical debt causes are: technological obsolescence, development environment changes, rapid commercial success and advent of new and better technologies.

During the past decade, technical debt was deeply studied and analysed along every software development life cycle process. In 2014 Alves et al. fathomed all the available literature related to Technical Debt and classified all its forms in the following ontology [17]:

- *Architecture debt*: issues in the project’s architecture (e.g., violation of modularity) that affect some architectural requirements (e.g., performance, robustness). It usually cannot be repaid only through source code interventions but it implies more extensive corrective development activities;
- *Build debt*: issues that make task building more time and processing consuming and harder than the necessary (e.g., unnecessary code to the customer);
- *Code debt*: bad coding practices in the source code that impact on its maintainability (e.g., reducing its legibility);
- *Defect debt*: known software defects whose fix are deferred to the future due to different priorities or limited resources;
- *Design debt:* bad design practices that violate the principles of good object-oriented design;
- *Documentation debt*: missing, inadequate or incomplete project documentation;
- *Infrastructure debt*: software organisation issues that can delay or hinter some development activities (e.g., infrastructure fix);
- *People debt*: people issues that can delay or hinder some development activities (e.g., new knowledge brokers);
- *Processes debt:* issues caused by inefficient processes;
- *Requirement debt*: trade-off between the requirements that a development team has to implement and how they implement them (e.g., requirements implement for a limited number of cases);
- *Service debt*: issues introduced by an inefficient web service substitution;

- *Test automation debt*: unnecessary work done by automated tests of previously developed functionality to support continuous integration and faster development cycles;
- *Test debt*: issues in testing activities that influence testing qualities (e.g. low code coverage).

Brown et al. defined the concept of "*anti-pattern*" as a "commonly occurring solution that will always generate negative consequences when it is applied to a recurring problem" [23], thus an anti-pattern is a pattern which implies negative connotations.

Within the Technical Debt research area, Fowler [36] defined the term "*Code Smell*" to refer to code patterns that can be symptoms of poor design and implementation choices. Code smells are usually considered as symptoms of the presence of anti-patterns and thus are mined to detect them. Since Code Smells can be characterised by sub-optimal development choices or they can be associated to some poor recurring design and implementation decisions, they can diminish code comprehension and increase change and fault proneness of a project. *Code Smells can be used as indicators of the presence of accumulated Technical Debt* [60].

Tamburri et al. analysed another type of debt in which a software development may incur, generated by non-optimal socio-technical decisions. This "*Social Debt*" is correlated to the social components of an organisation and it was defined as the "*unforeseen project cost connected to a sub-optimal development community*" [68]. Social Debt was later redefined in 2015 by the same authors as the "cumulative and increasing cost in the current state of things, connected to invisible and negative effects within a development community" [70].

While decisions in Technical Debt are about technologies and their applications, *decisions that cause Social Debt are about social interactions and people themselves.* Social Debt shares many aspects with Technical Debt since they have many similarities and common points. Social Debt, as well as Technical Debt, can be used as an indicator of the development process quality, considered as the result of past accumulated decisions [70]. Tamburri et al. highlighted this relation between the two diametrically opposite typologies of debt paraphrasing the Cunningham's definition of technical debt and describing Social Debt as "not quite right development community - which we postpone making right" [70].

Global Software development is characterised by many socio-technical decisions (e.g., outsourcing, organisational structure, communications organisation) that do not only influence the technical area but even the social one, influencing how people interact, communicate and organise themselves. Since socio-technical decisions can influence and modify people's social behaviours, in addition to Technical Debt, they may produce Social Debt due to non-optimal socio-technical decisions. Social and Technical Debt can generate delays and addictions costs within the development process or within the development community, that may increase over time and be invisible or intentionally delayed due to the intrinsic nature of social and technical debt.

De Farias Junior et al. [32] conducted a study on communication related risks in distributed software development that can be considered a Social Debt study because it analyses some organisational issues created by a non-optimal usage of communications within a software company. They considered as communication related risks: issues related to physical and temporal distance, trust, difference of cultural and linguistic orientations between different teams. To avoid or mitigate the listed communication related risks within a distributed software development, Farias Junior et al. proposed these recommendations:

- *encourage frequent communication*: it reduces misunderstandings created by cultural and linguistic differences and it increases distributed teams cohesion and trust, which can generate an increment informal communications between developers;
- *establish an appropriate communication infrastructure*: it addresses uncertainty and unpredictability of the communications and it reduces the negative effect of the absence of "face-to-face" meeting;
- *promote socialisation*: it increases cohesion, inter-personal relationships between different team members, communication effectiveness and informal communication;
- *encourage effective communication*;
- *promote visits among distributed sites*: it increases trust and it constitutes new interpersonal relationships with the creation of new informal communications;
- *promote informal communication*: it diminishes misunderstandings, it creates trust and facilitates knowledge sharing;
- *promote cultural awarenes*s and adopt group-ware applications.

In Social Debt studies, mirroring the Code Smell concept, it is possible to define "*Community Smells*" as social related anti-patterns useful to understand negative community characteristics and trends. *Community Smells are formally defined as "socio-technical anti-patterns that may appear normal but in fact reflect unlikeable community characteristics"* [70], thus Community Smells identify anti-social organisational behaviours within a community. An example of Social Debt is when developers refuse or delay information sharing for any reason. Community Smells are a set of social and organisational circumstances with implicit causal relations which do not constitute a problem if considered alone but that if repeated over time, they may cause Social Debt in the form of delays, mistrust, uninformed and miscommunicated architectural decision-making.

Social Debt, as its technical counterpart, can be paid back adopting specific socio-technical decisions with the purpose of mitigating a precise Social Debt aspect, possibly detected by a Community Smell. Tamburri et al. [70] found some "*mitigations*" that were proven to have a beneficial effect on Social Debt reduction and discovered that some socio-technical decisions made to extinguish contracted Social Debt eventually worsen the situation or did not yield positive outcomes (40% of socio-technical mitigations considered). Mitigations addressed to resolve Community Smells and to pay back the related Social Debt are called "deodorants".

Architectural decisions were considered in both Technical Debt [35] and Social Debt [67] studies and in both cases they are highlighted as one of the most important cause of debt generation in professional software environments. Ernst et al. [35] studied the relation between Technical Debt and architectural decisions and they concluded that architectural issues are the most relevant cause of technical debt generation and that to pay back such generated Technical Debt is hard because usually the incriminated architectural decisions were taken many years in the past. Tamburri et al. [67] further investigated architectural decisions with a Social Debt perspective, identifying architectural smells and proposing a possible metric to measure potential Social Debt contracted in software architecture processes. Their identification methodology, based of Social Network Analysis theories, computes the communicability of an architectural design decision to identify architectural smells with the purpose of avoiding or diminish the related side effects (e.g., architecture erosion, lack of vision, mistrust).

Referring to Conway's law and its related studies, it is possible to re-conduct Technical Debt to not-optimal development processes decisions and Social Debt can be re-conducted to not-optimal organisational processes. Since socio-technical decisions are indirectly correlated to Social Debt [70] and that socio-technical congruence can be considered an agreement indicator to Conway's law, *socio-technical congruence can be considered as a metric to identify possible Social Debt present within a community because it quantifies the similarity of social and technical processes whenever a communication need is present.*

Both Social or Technical Debt can depend from the context evolution because it is possible that the original decision which created it, was correct but as time passed the context changed and such decision was not positive in retrospect.

In analogy to the monetary debt, Technical or Social Debt in software engineering is not necessary a bad thing if it is known, accepted and controlled. For example sometimes Technical Debt is necessary to move forward the project development. Debt, similarly to congruence [64], may imply trade-offs because resolving the debt in a particular level may create another debt in another level. Potdat et al. [62]

discovered that self-admitted Technical Debt in Open Software development is a common phenomenon (from 2.4% to 31% of project files is affected), that developers with higher experience usually tend to introduce most of the self-admitted Technical Debt and that in the optimal case only slightly more than half of the introduced debt is paid off.

In their Technical Debt ontology published in 2014, Alves et al. [17] identified a Technical Debt category called "people debt" that can be associated to the concept of Social Debt. They define people debt as the debt associated to people issues, in the context of software organisation, which may delay or hinder development activities and they provide as an example the case of a concentration of expertise limited to few people as a consequence of delayed training and/or hiring.

Another study that can be considered belonging to the Social Debt research area is the one conducted in 2014 by Zhou et al. [76], concerning the quantification of global team performance and profitability, because it investigated the effects of some social and organisational structure properties (e.g., temporal dispersion, language difference, skilled workers turnover). Zhou et al. state that "the right organisational structure is required to achieve benefits of lower labour costs", acknowledging the fact that "combined effects of the [social and organisational] factors could lead to reduced profitability" and concluded, in complete accord to Social Debt definition, that "in some extreme cases the cumulative effects of external factors could outweigh the advantage of lower labour rates for globally outsourced work".

In "Why good developers write bad code", published in 2015, Lavallée et al [46] analysed the relationships between some organisational factors and their impact on developers' working conditions and performances. Lavallée et al. identified the following socio-technical organisational issues that may compromise the software quality and its success [46]:

- *Documenting complexity*: presence of large, old and poor documentation that causes the unwanted situation in which unimplemented requirements are discovered at the end of software development. This issue is related to knowledge management and support maintenance activity transmissions;
- *Internal dependencies*: presence of many inter-dependent modules that create conflicts between projects on the deployment schedule;
- *External dependencies*: changes to third-party modules result in costly delays;
- *Cloud storage*: third-parties do not support vulnerability testing;
- *Organically grown processes:* software defects are documented but developers are not aware of them because information exchange is hindered by frontiers between processes. This issue is caused by the creation of "islands of for-

mality", which are zones where different processes have limited interactions between them;

- *Budget protection*: due to the external dependencies issue an "home-made patch" approach, through the creation of wrappers, is preferred to avoid additional third-party costs;
- *Scope protection*: teams try to deny every other team's change requests to protect their project's scope;
- *Organisational polities*: issues that arise when the wrong or uninformed person is contacted;
- *Human resource planning* (truck number [30]): development is performed in silos and there is a high possibility of project knowledge loss due to developer turnover or delays due to developer's unavailability;
- *Undue pressure*: developers are threatened to deliver in time by managers and senior developers.

Comparing this research results to Tamburri et al. researches about Social Debt, it evident that there are many common findings and shared results, especially considering the more "human-related" socio-technical issues: "organically grown processes", "scope protection" "organisational polities" and "human resource planning". Lavallée et al. study, conducted within a large commercial company, can be considered in every aspect a Social Debt related research because the authors conclude their work stating that software quality can be negatively affected by decisions taken under certain organisational conditions and assert that "the design flaws introduced because of the organisational issues presented here will no doubt come back to haunt" [46], which it is a paraphrase of the Social Debt definition proposed by Tamburri et al. [70].

2.6 Community Smells

An empirical software engineering research paper that constituted a fundamental contribution to the Social Debt research area was conducted by Tamburri et al. and it was named "Social Debt in software engineering: insights from industry" [70]. Tamburri and his collaborators improved the Social Debt definition, added background to its insurgence's conditions, highlighted possible mitigations to Social Debt and provided the precious contribution of defining several Community Smells that were proven to be capable of detecting the presence of Social Debt. Tamburri et al. analysed correlations between a set of socio-organisational circumstances and the raise of additional costs in software processes within a large industrial software case study. They summarised the identified Social Debt circumstances in a framework, relating them with their causes, consequences, conditions, contexts, covariances, contingents, anti-patterns (“Community Smells”) and they suggest some possible techniques useful to avoid the insurgency of such negative circumstances. In their research Tamburri et al. were able to capture some Social Debt characteristics [70]:

- Socio-technical decisions and Social Debt are indirectly connected;
- Social Debt is an emergent property of the development community itself and, despite Technical Debt, it cannot be ascribed to any particular software artefact or operation in the development process but, at the same time, Social Debt has a strong effect on many different software artefacts;
- Social anti-patterns (i.e., Community Smells) can be considered as indicators of the emergence of Social Debt within a community;
- Social Network Analysis methodologies can be used to identify Community Smells and quantify Social Debt costs;
- Social Debt can generate Technical Debt;
- Specific socio-technical decisions (“mitigations”) can be implemented to pay back totally or partially the detected Social Debt.

It is possible to classify Community Smells within three different classes: smells related to the community structure and its related properties (e.g, community formality), smells related to the community context (e.g., political boundaries) and smells related to the community members’ interactions (e.g., socio-technical relationships). In their industrial case study, Tamburri et al., identified and classified nine different Community Smells [70]:

1. **Organisational Silo Effect**: this Community Smell occurs when it is present a too high decoupling between developers and their related development tasks. This occurrence causes low mutual awareness, low socio-technical congruence and lack of communications and cooperation in checking task dependencies within the community. An organisational silo is present in the development community whenever an isolated subgroup of loosely dependent development partners waste resources (e.g., time) or duplicate resources over the development life-cycle. Another possible side effect of this Community Smell can be the establishment of a “tunnel vision” due to the lack of cooperation and collaboration, which may imply a lack of creativity within the development team and eventually developers will make architectural decisions without the necessary authority and knowledge. A mitigation to the Organisational Silo Effect Community Smell is the institution of a “social wiki” within the development

community, combining developers profiles with the artefacts they are working on and the related documentation.

2. **Black-cloud Effect**: this Community Smell occurs when two concurrent circumstances take place together: lack of people able to cover the experience or knowledge gap between two software products and the lack of official and periodic knowledge sharing opportunities (e.g., daily stand-ups). Whenever those two circumstances are verified, every knowledge exchange initiative can create a confusing communication overload ("black-cloud") with back-and-forth emails which obfuscate reality. This Community Smells is caused by the absence of officially defined sharing protocols, lack of boundary spanners (individuals whom link internal team network to other teams) and presence of not efficient information filtering protocols. The main side effects of this Community Smell are the creation of mistrust between developers, the possibility of information obfuscation and the rise of egoistic behaviours (e.g., developers take decisions even if they do not have decisional authority). As for the Organisational Silo Effect Community Smell, the adoption of a "social wiki" can mitigate the negative effects of the Black-cloud Effect Community Smell.

3. **Prima-donnas Effect**: this Community Smell occurs whenever a team of developers is unreceptive to change its internal processes and/or characteristics, or it is unwilling to be influenced by external team members through the forms of collaborations and/or communications. This selfish and condescending team behaviour can create serious isolation problems and tensions between the community and the "prima-donnas" team, which is unable to welcome support from other development partners. Prima-donnas Effect Community Smell can raise due to stagnant collaboration within the community, due to inefficient structural innovation or due to organisational inertia. The consequences of the lack of collaboration and communication can be worsened by organisational changes because they create fear in the prima-donnas teams and increase their egoistical behaviours. A possible mitigation for the Prima-donnas Effect Community Smell is the institution of "culture conveyors", which allow an harmonization process of different organisational cultures through the promotion of developers coming from different communities to the role of architects. Another mitigation technique is the adoption of a "community-based contingency planning" in which managers decide to make technical and socio-technical decisions together and use the learning community as a device to generate contingency plans, if some decisions lead to undesirable outcomes. As for the Organisational Silo Effect and Black-cloud Effect Community Smells, the adoption of a "social wiki" can mitigate the negative effects of the Prima-donnas Effect Community Smell.

4. **Leftover-techie Effect**: this Community Smell is caused by an increasing isolation of the maintenance and the help-desk operations from the operative people, with a related feeling of abandonment by the technicians. The main side effect of this Community Smell are mistrust and the emergence of a sharing villainy behaviour, related to knowledge and status awareness. A mitigation capable of reducing Social Debt connected to this Community Smell is the "full-circle" and it consists in the creation of a dedicated communication line (e.g., instant-messaging) between key developers, managers and operation technicians.

5. **Sharing Villainy**: this Community Smell is caused by the absence of experience sharing initiative and high-quality knowledge exchange activities (e.g., face-to-face meetings), in addition to a shared mindset that considers knowledge interactions between developers as wasting time activities rather than positive opportunities. The main side effect of the Sharing Villainy Community Smell is the limitation to developers' propensity to share knowledge and meaningful experiences, to the extreme of sharing outdated, wrong or unconfirmed information. A possible mitigation technique for this Community Smell, as for the Prima-donnas Effect Community Smell, is the creation of "culture conveyors". As for the Organisational Silo Effect, Black-cloud Effect and Prima-donnas Effect Community Smells, the adoption of a "social wiki" can mitigate the negative effects of the Sharing Villainy Community Smell.

6. **Organisational Skirmish**: this Community Smells occurs whenever operations and development units are misaligned in their organisational culture (e.g., organisational layout and properties), in their communication habits and in their expertise levels. These misalignments cause severe managerial issues and delays.

7. **Architecture Hood Effect**: this Community Smell occurs whenever decision-makers are not well integrated and geographically distant from other developers and operators of the community and their decisions are taken using a software architects board that makes decisions' responsibility and logic not easily discernible. The side effects of this Community Smell are the inability to identify decision-maker responsibilities and the unwillingness of developers to accept decisions with a related uncooperative behaviour within the development community. Architecture-hood Community Smell can be mitigated by the socio-technical decision of adopting "stand-up voting" in an anonymous form to accept decisions at the end of fixed daily stand-ups.

8. **Solution Defiance**: this Community Smell occurs when the development community divides itself into overly similar subgroups with different levels

of cultural and experience backgrounds; those homophile subgroups divide themselves into smaller factions with opposite and conflicting opinions toward some socio-technical decisions that should be taken. The side effects of this Community Smell are delays, uncooperative behaviours, decisions ignoring and "organisational rebellion" due to the unwillingness of developers to take a shared decision within different factions until the very last possible moment. A socio-technical decision to mitigate the effect of solution defiance, as for the Prima-donnas Effect Community Smell, is the adoption of a "community-based contingency planning". As for the Organisational Silo Effect, Black-cloud Effect, Prima-donnas Effect and Sharing Villainy Community Smells, the adoption of a "social wiki" can mitigate the negative effects of the Solution Defiance Community Smell.

9. **Radio Silence**: this Community Smell occurs when the organisational structure is highly formal, complex and constituted by regular procedures which cause changes to be delayed and people time to be wasted due to required formal actions and filters hiding necessary information. The main side effect of this Community Smell is the massive delay in decision making processes due to people unavailability or due to further information needs. A mitigation able to reduce almost completely the negative effects of the Radio Silence Community Smell is the creation of a "learning community" that involves all the developers and operators. This socio-technical decision can reduce delays in a direct way with the creation of strong organisational and social relationships between developers and in an indirect way, enabling a passive knowledge sharing channel.

Another important contribution to the Community Smell area, from the operationalisation point of view, comes from Magnoni work [50]. In his master thesis Magnoni has developed an extension to Codeface, an Open Source "framework and interactive web front-end for the social and technical analysis of software development projects" [6], this extension ("Codeface4Smells") offers a lens to observe software development communities from a quality perspective and diagnose organisational issues in an automated tool-supported fashion. Magnoni's work is heavily based on Tamburri's et al. industrial case study [70] and is able to automatically detect four out of nine Community Smells identified in [70]:

1. **Organisational Silo Effect:** while working on possible automatic identification pattern of this Community Smell Magnoni focused on the most important Organisational Silo Effect side effects: decrease of communications within the community and generation of a "tunnel vision"; therefore he proposed two different identification patterns in order to provide the ability of identifying both typologies of side effects:

   (a) **Organisational Silo Effect**: detection of *community members who collaborate with other members but who do not communicate* within the analysed communication channel;

   (b) **Missing Links**: detection of *development collaborations between two community members that do not have communication counterparts*;

2. **Black-cloud Effect**: detection of *isolated sub-communities that, in different and subsequent time periods, do not communicate with the exception of one communication link*;

3. **Prima-donnas Effect**: detection of *isolated sub-communities that cooperate on similar parts of the source code but do not communicate with the exception of one communication link*;

4. **Radio Silence**: detection of *unique knowledge and information brokers toward different sub-communities*;

Chapter 3

Problem analysis

This chapter summarises the fundamental aspects of this master thesis and provides an overview of how our empirical software engineering research was defined, characterised and executed.

Section 3.1 presents the definitions of a set of key terminologies that should be kept in mind while reading this work. The motivations at the root of this master thesis and the research questions that were addressed within the execution of this empirical software engineering study are described in Section 3.2. The original contributions constituting the basic building blocks of this master thesis are presented in Section 3.3, together with their characteristics, usefulness and purposes. Finally, Section 3.4 explores the context of the dataset considered within this research, motivating and providing information about the set of Open Source Software development communities considered in the analysis.

3.1 Definitions

This section provides definitions of several important and fundamental terminologies used within this master thesis.

- **Factor**: element, circumstance or influence which contribute to produce a result;
- **Socio-technical factors**: elements, circumstances or influences which contribute to produce a result that has both social and technological aspects;
- **Technical Debt**: Unforeseen project costs connected to a "sub-optimal" development decisions and executions [31];
- **Code Smells**: Development anti-patterns that produce negative effects on the long run and lead eventually to Technical Debt [36]. Therefore, Code Smells represent a risk correlated to the potential presence of Technical Debt;

- **Social Debt**: Unforeseen project costs connected to a “sub-optimal” development community habits [68];
- **Community Smells**: organisational and social patterns that produce negative effects on the long run and lead eventually to Social Debt [70]. Therefore, Community Smells represent a risk correlated to the potential presence of Social Debt;

3.2 Research questions

The main goal of this master thesis is the *discovering and the understanding of any possible correlations between Code and Community Smells in the context of open source development communities.*

In order to achieve our goal we performed an empirical *software engineering research*, aimed to discover and understand the possible existence of Community and Code Smells correlations within an open source community.

This master thesis aims at addressing the following research question:

- **RQ1.** *Does a correlation between Community and Code Smells exists?*

  Considering the literature, the following sub-questions were formulated:

  - **RQ1a.** *How strong this correlation is?*
  - **RQ1b.** *Does this correlation changes between different Community Smells?*
  - **RQ1c.** *Is it possible to discover fine grained association between a Community Smell and the developers involved?*
  - **RQ1d.** *Is it possible to detect Code Smells in a time line fashion?*
  - **RQ1e.** *Is it possible to approach the entire problem in an automated way?*

3.3 Contributions

This section briefly introduces the main basic building blocks on which this master thesis is founded. Each of them can be considered a precious *original contribution* to the Social Debt and socio-technical research fields as they provide additional information, methodologies and tools.

The main contributions of this master thesis are:

- **Complete analysis automation.** We have built a series of software automated procedures able to perform a full Codeface4Smells analysis in a complete unattended fashion. These procedures are in charge of gathering all Codeface4Smells prerequisites and execute a complete analysis run;

- **Granular association of Community Smells to single developers.** We have slightly modified the deep heart of Codeface4Smells' Community Smells detection mechanism in order to unroll previously aggregate data to the granularity of single developer that is responsible for the smell;
- **Time-windowed Code Smells detection.** We have added a new analysis step to Codeface4Smells in charge of static Code Smells detection and collection. This new step deeply exploit previously generated time series data, git "time machine" ability and a static code analysis tool by Palomba [59];
- **Association between Code and Community Smells.** Within the context of the new Codeface4Smells analysis step, we have added the ability to link Code and Community Smells. By exploiting existing and newly produced data and with the help of some simple but effective SQL manipulation we have finally been able to create that link.

The interaction work-flow between the previously introduced basic building blocks follow the same order in which they have been reported; the sole exception to the previous sentence is analysis automation block, which indeed is the very first block the work-flow "hit", but its effect are somehow "surrounding" and present for the entire analysis work-flow. All the following steps (can) occurs in the context of a single Codeface4Smells analysis run.

1. Optional, but strongly advised. Start an automated analysis and the software will take care of everything, starting from the prerequisites (i.e. git repository, mailing list, configurations files, etc.) to the actual step-by-step analysis run. This functionality is available only for the Codeface4Smells "know projects" as better explained in Section 4.3.1;
2. In the context of Codeface4Smells' Community Smells detection algorithms, we have introduced a sort of unroll capability in order to get and store smells raw data for further elaborations;
3. We have then introduced a brand new analysis step whose aim is to detect and collect static Code Smells data. All data collected in both this and the previous work-flow steps are the basis for the elaborations that will occurs in the following steps;
4. Data produced in the previous steps are merged and manipulated with the help of some simple SQL transformations in order to produce a new dataset containing all the links between Community and Code Smells with the level of detail that is required for the following work-flow step.

#	Project	Institutional Website	Source code repository	Development mailing list (Gmane)
1	Cassandra	https://cassandra.apache.org/	http://git-wip-us.apache.org/repos/asf/cassandra.git	gmane.comp.db.cassandra.devel
2	Cayenne	https://cayenne.apache.org/	git://git.apache.org/cayenne.git	gmane.comp.java.cayenne.devel
3	Jena	https://jena.apache.org/	git://git.apache.org/jena.git	gmane.comp.apache.jena.devel
4	Mahout	https://mahout.apache.org/	https://github.com/apache/mahout.git	gmane.comp.apache.mahout.devel
5	Tomcat	https://tomcat.apache.org/	git://git.apache.org/tomcat.git	gmane.comp.jakarta.tomcat.devel
6	Ant	https://ant.apache.org/	https://github.com/apache/ant.git	gmane.comp.jakarta.ant.devel
7	POI	https://poi.apache.org/	https://github.com/apache/poi.git	gmane.comp.jakarta.poi.devel
8	Scala	https://www.scala-lang.org/	https://github.com/scala/scala.git	gmane.comp.lang.scala
9	Eclipse CDT	https://eclipse.org/cdt/	https://git.eclipse.org/r/cdt/org.eclipse.cdt	gmane.comp.ide.eclipse.cdt.devel
10	Jackrabbit	https://jackrabbit.apache.org/jcr/	git://git.apache.org/jackrabbit.git	gmane.comp.apache.jackrabbit.devel

Table 3.1: List of analysed projects

3.4 Dataset selection

The dataset considered in this master thesis consisted of ten Open Source Software projects. The complete list of analysed projects can be consulted in Table 3.1, where for every considered project it is showed its name, its institutional website, its code repository address and the development mailing list considered as its primary development communication channel. As it is possible to deduce from the list of analysed projects, we considered FLOSS development communities of different dimensions, popularity, development habits, openness and application contexts.

Our choice of FLOSS development communities was not random but it was guided by specific and rigid requirements, dictated by the infrastructure of Codeface4Smells or by analysis requirements. More specifically, it was mandatory that every analysed project was characterised by the following list of must have requirements:

1. Source code is available on-line through git repositories;
2. Project programming language is Java;
3. It was possible to identify an ≪active≫ development mailing list and its archive was present on `www.gmane.com`;
4. Within every analysed window it was sent at least one e-mail to the considered development mailing list;
5. Within every analysed window it was committed at least one source code contribution;
6. Codeface4Smells collaboration, communication and socio-technical analysis terminated without errors.

The list of analysed projects were partially retrieved from datasets used in previous empirical software engineering researches [50, 33], in which diversity was verified with respect to several factors.

Every FLOSS project was analysed executing Codeface4Smells analysis using three-month analysis windows for the last three years, therefore every project's analysis was constituted by a total of 12 ranges. We used this temporal analysis window because it was previously used in another empirical software engineering researches based on the usage of both Codeface [43] and Codeface4Smells [51] and because it was demonstrated that a software development community does not change significantly after a consideration window of three months [53]. The total time lapse of three years was set in order to achieve a relevant number of analysed ranges capable of capturing trends and correlations. Another motivation to the decision of limiting the analysis to three years is that before that temporal limit, the standard within FLOSS development was Centralised VCS, as explained in Section 2.4, and thus information about the author of the software contributions were not accessible.

Chapter 4

Codeface4Smells Enhancement

This chapter presents contributions of this master thesis to the original Codeface4Smells architecture and functionalities.

Section 4.1 presents Codeface and its macro architectural functional block, original Codeface4Smells architecture and peculiar capabilities are described in Section Section 4.2 and finally in Section Section 4.3 all contributions and enhancements to Codeface4Smells introduced in this master thesis are explored.

4.1 Codeface

Codeface is an Open Source "framework and interactive web front-end for the social and technical analysis of software development projects" [6], which is capable to retrieve and analyse collaboration and communication relationships of a software development community using different software development artefacts (Version Control Systems and mailing lists). Codeface was created in 2010 by Wolfgang Mauerer and most of its development is internally executed and sponsored by Siemens. It is written mainly using python and R and it is released under the GNU General Public License v2.0.

Codeface analysis results can be useful to learn more about an embedded software ecosystem and all the retrieved information about a software project may help to understand and exploit collaboration and communication patterns and characteristics, highlight development issues and assist maintainers in project control and management activities.

The software architecture of Codeface, as showed in Figure 4.1, is constituted by the following layers:

1. *Common layer*: constituted by common routines, SQL abstraction, projects configuration and logging functionalities;

2. *Version Control Systems analysis*: computes evolutionary project metrics (*Tim-*

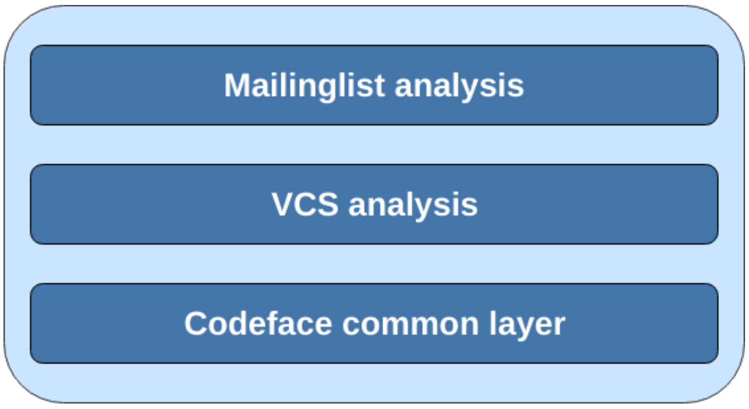


Figure 4.1: Architecture of Codeface

ing analysis), creates the developer network and execute cluster analysis (*Collaboration analysis*)

3. *Mailing lists analysis*: performs communication analysis.

Codeface constitute the very basis of our work and has been further expanded with the Automation layer as better explained in Section 4.3.1.

4.2 Codeface4Smells

Codeface4Smells is build on top of Codeface's software source code, thus it can be considered as an extension of Codeface, and it has the purpose of introducing software enhancements and new socio-technical analysis and Community Smells detection capabilities. Codeface4Smells is a brand new software layer, which, as showed in Figure 4.2, resides on top of all the already present Codeface architecture layers, and its socio-technical analysis capabilities are enabled by Codeface's communication and collaboration analysis outputs.

Codeface4Smells peculiar capabilities are the followings:

1. **Creation of a global Developer Social Network**, generated from the combination of communication and collaboration Developer Social Networks;

2. **Introduction of automatic ranges detection** to analyse at most the last three years of a project considering three months windows;

3. **Identification of sub-communities** within the global, communication and collaboration Developer Social Networks;

4. **Identification of core developers** within the global, communication and collaboration Developer Social Networks;

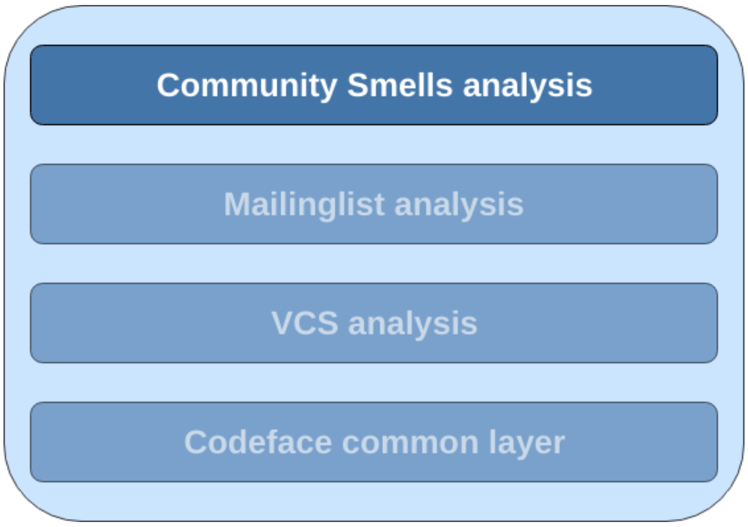


Figure 4.2: Architecture of Codeface4Smells

5. **Identification of developers supported by commercial companies** or self-employed within a project development community;

6. **Introduction of Socio-technical Quality Framework** to gather more information about socio-technical characteristics of a software project development environment and its community;

7. **Detection and quantification of Community Smells** which may indicate the presence of Social Debt within the project development;

8. **Correlation analysis** (Pearson and Spearman) execution between socio-technical quality factors and the occurrence of Community Smells;

9. **Reports and graphs** generation to simplify access to socio-technical and Community Smells analysis results.

Codeface4Smells, mainly its detection and quantification of Community Smells, had been enhanced in the context of this master thesis with the capability of granular association between Community Smells and the "guilty" developer as better explained in Section 4.3.2.

4.3 Enhancement

Codeface4Smells architecture, as shown in Figure 4.3, has been enhanced with four main contributions. *Automated analysis* better explained in Section 4.3.1, *Community Smell Granular analysis* capability better explained in Section 4.3.2, *Technical analysis* better explained in Section 4.3.3, *other general enhancement* are described in Section 4.3.4.

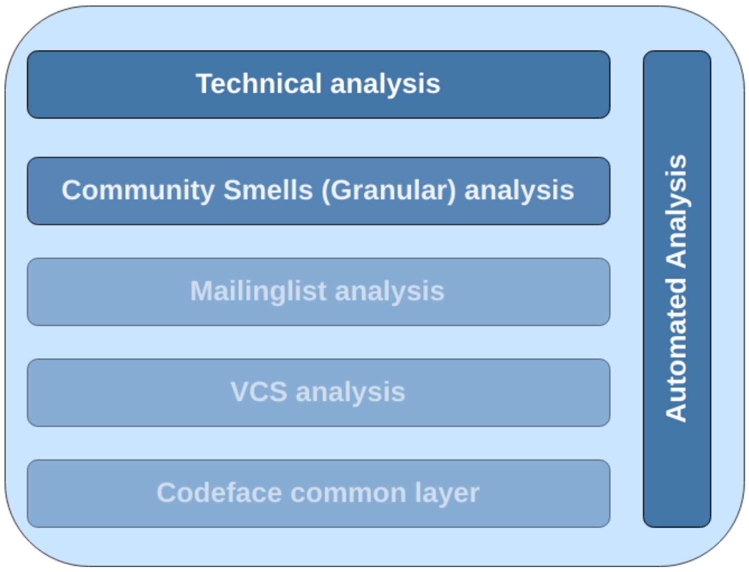


Figure 4.3: Architecture of Codeface4Smells with Technical analysis

In the following Sections we are going to better and deeper explain all the previously introduced enhancements.

4.3.1 Automated analysis

Automated analysis, as shown in Figure 4.3, is a step that cross the entire Codeface4Smells analysis run; its aim is to help the final user to go through a complete project analysis and have all the data and hints without worrying about all the tedious prerequisite data collection.

Automated analysis internal structure is divided in the following main components:

- **Known projects database**. This database contains the list of all projects used as base dataset for this master thesis. The database is available at "*$project-root-folder$*/known-projects/db.csv" and is saved using the standard international csv (comma separated values) format. The database has the following fields structure:
  - *Project*: represents the project unique name. This field is mandatory;
  - *Official Website*: it's a link to the institutional project website. This field is optional;
  - *Repo*: it's the actual git repository address to be used to download all project source code files. This field is mandatory;
  - *Gmane Mailing list*: it's the actual name of project's mailing list repository in Gmane collection. This field is mandatory.

- **Prerequisite gatherer**. This component was created to make life of Codeface4Smells final user easier. Codeface4Smells, in order to successfully complete an analysis run, heavily rely on a lot of conventions starting from folder structure going to files names to which the prerequisites must adhere, this component, ultimately, was developed to hide all this complexities to the final user. Some of the tasks which the component is in charge of are summarised in the following list:
  - *Creation of folder structure*: all the output artefacts of Codeface4Smells are placed inside a rigid folder structure, its creation was in charge of the final user;
  - *Creation of analysis steps config file and parameters*: as for the folder structure, each analysis step requires a set of configuration parameters, some of them must be stored in particular files. Creation of these files and the choice of parameters value was totally in charge of the final user;
  - *Source code and mailing list download.*
- **Mailing list download and clean**. This part had been the trickiest one mainly because in June 2016 Gmane had been turned off by the maintainer, but, luckily for us, the NNTP bridge of gmane had been acquired and turned on by new maintainers. After the maintainers switch all the web api, that were used by the original implementation of Codeface for mailing list statistics, stayed offline therefore we were forced to re-implement the mailing list download procedure in order to overtake this api lack. The cleaning part, that was mandatory since in some mailing lists many Asiatic or non-common characters were used, was already part of Codeface4Smells therefore we just integrated the existing code in our architecture;
- **Command Line Interface "smells" command**. This component added a new option to the existing Codeface command line interface (CLI), the "smells" option. Once called with the "smells" option, Codeface4Smells starts an entirely automated analysis process. The "smells" option is in charge of managing the life-cycle of Codeface's identity service (a task that was previously in charge of the user), automate the analysis steps invocation and is also able to remember the last successfully executed step in order to let the user interrupt and then restart the analysis at the same point in the process.

Even if this contribution could be seen as secondary, or at least not so significant, it helped us a lot because it has simplified the entire analysis process and relief us from all those error prone tasks that we have previously described.

To summarise, the values introduced with these contributions are:

- Overall analysis automation process can be easily extended with very few modifications to the new “smells” CLI option;
- Pre-requisites gathering is not anymore in charge of the final user;
- Final users can either start or resume an analysis with the same command;
- Final users are able to start brand new project analysis by just filling all the mandatory data in the “known-project” database and everything else is managed by Codeface4Smells.

4.3.2 Community Smells Granular analysis

Community Smells Granular analysis, as shown in Figure 4.3, is an enhancement of Community Smells analysis block present in Figure 4.2; modifications to Codeface4Smells original code are indeed not so big, but very significant for the analysis steps that comes after this one.

Granular analysis main modifications and contributions are the following:

- **Data model extension**. We have extended the original data model in order to save Community Smells unrolled raw data. Original Codeface and Codeface4Smells data model were stored using MySQL DBMS, our “Granular” extensions include the tables shown in Figure 4.4.
    - Table *sociotechnical* contains all the data coming from original Community Smells analysis, previously those data were stored just in a csv file inside the analysis result folder. We have introduced this table in order to have those data saved in one place with others analysis data to unlock the possibility to use them for further analysis. Table structure and columns, with the exception of columns *id* and *releaseRangeId*, follows the representation and meaning described in [50].
    - Table *sociotechnical_smells* contains a mere catalogue of all Community Smells that Codeface4Smells is able to detect. This table have been created mainly for reporting reasons. Table structure is the following:
        * *id*: Community Smell unique identifier;
        * *name*: Community Smell name.
    - Table *sociotechnical_granular* contains the actual granular unrolled raw data about Community Smells detected by Codeface4Smells. Data in this table act as a fundamental basis for all work made in Technical analysis step. Each row instance represent the occurrence of a Community Smell and clearly identify the exact moment in time (using a release time span) in which it occurred and point out the “guilty” developer responsible for the smell. Table structure is the following:

* *id*: row unique identifier;
* *releaseRangeId*: reference key to the exact moment in time;
* *socioTechnicalSmellId*: reference key to the Community Smell;
* *personId*: reference key to the “guilty” developer.

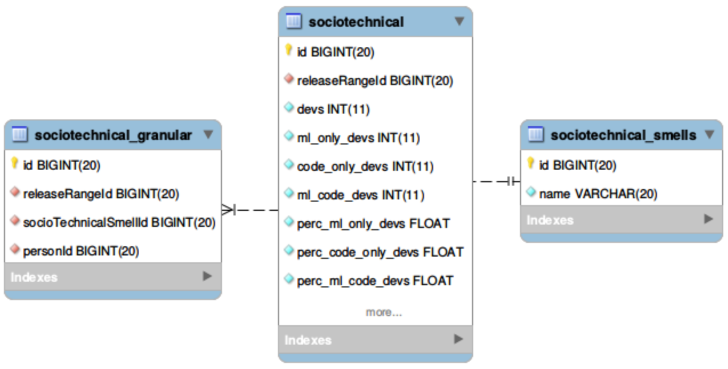


Figure 4.4: Data model extensions

- **Data unrolling mechanism**. The algorithm we have built, thanks to the power of R programming language and the way Community Smells are detected, is pretty straightforward. The algorithm is deeply integrated inside the existing Community Smells detection, the flow is the same for each analysed release:

  1. Transform Community Smell data from tabular into vector form;
  2. Apply specific filter on the vector from step 1. In case of *Organisational Silo* smell (see [50]) take only element with odd index;
  3. Delete duplicate value;
  4. Create a per-release collection with all the detected and processed smells;
  5. Persist the collection into *sociotechnical_granular* table.

To summarise, the value introduced with these contributions are:

- Is now possible to have raw Community Smells data at your disposal;
- The newly introduced data model is quite generic and can easily store any kind of new Community Smell;
- The computational overhead introduced by both data unrolling and data saving is almost irrelevant.

4.3.3 Technical analysis

This phase, as shown in Figure 4.3, represents the last analysis step introduced in this renewed Codeface4Smells architecture. In order to avoid the writing of yet another static code analysis tool we have used an experimental tool from Palomba [59].

Technical analysis main modifications and contributions are the following:

- **Command Line Interface "tsa" command**. This component added a new option to the existing Codeface command line interface (CLI), the "tsa" option. The "tsa" option is in charge of managing the Technical analysis process step. The "tsa" command is designed to be the last in the process analysis run, therefore all previous steps must have successfully completed their execution;

- **RunCodeSmellDetection.jar**. This Java executable layer has been added over the Palomba's experimental tool. Our contribution to this tools is a new layer on top of the actual Code Smells detector that is able to scan all files in an entire repository and create a cumulative report of all affected files with their respective Smell. As a result we are now able to analyse a Java code repository and export in csv format the analysis result. Palomba's tool which is based on Moha et al. work [57] is able to identify the following Code Smells:

    - *Class Data Should Be Private* smell: occurs when a class has more then ten public instance variable;
    - *Complex Class* smell: occurs when a class has a McCabe Cyclo Complexity [52] greater then two hundreds;
    - *Functional Decomposition* smell: anti-pattern may occur if experienced procedural developers with little knowledge of object-orientation implement an object-oriented system;
    - *God Class* smell: corresponds to a large controller class that depends on data stored in surrounding data classes;
    - *Spaghetti Code* smell: is an anti-pattern that is characteristic of procedural thinking in object-oriented programming;
    - *Has Long Methods* smell: occurs when a class has one or more methods with more then one hundred Single Line of Code (SLoC) and more then two input parameters.

- **Data model extension**. As for Section 4.3.2 we have introduced new tables (Figure 4.5) and views (Figure 4.6) in order to store both partial and final analysis data.

- Table *techsmell* contains all raw static code analysis data coming from Palomba tool [59]. Each row instance means that at some point in time that class was affected by at least one of the available code smells. The actual code is shown in Algoritm 4.1. Table structure is the following:
    * *id*: row unique identifier;
    * *releaseRangeId*: reference key to the exact moment in time;
    * *className*: name of the class;
    * *classFilePath*: path of the class's source file inside the project repository;
    * *classDataShouldBePrivate*: boolean value, true if the class is affected by *Class Data Should Be Private* smell, false otherwise;
    * *complexClass*: boolean value, true if the class is affected by *Complex Class* smell, false otherwise;
    * *functionalDecomposition*: boolean value, true if the class is affected by *Functional Decomposition* smell, false otherwise;
    * *godClass*: boolean value, true if the class is affected by *God Class* smell, false otherwise;
    * *spaghettiCode*: boolean value, true if the class is affected by *Spaghetti Code* smell, false otherwise;
    * *hasLongMethods*: boolean value, true if the class is affected by *Has Long Methods* smell, false otherwise.
- Table *tech_and_community_smells* contains all Technical analysis final data produced by this analysis step. Each row instance means that a Community Smell "guilty" developer has modified, in a specific release, a class that is also affected by a Code Smell in the same release. Table structure is the following:
    * *id*: row unique identifier;
    * *releaseRangeId*: reference key to the exact moment in time;
    * *tag*: git commit hash specific of the release;
    * *authorId*: reference key to the commit author;
    * *socioTechnicalSmellId*: reference key to the Community Smell;
    * *smellName*: Community Smell name;
    * *file*: path of the class's source file inside the project repository;
    * *classDataShouldBePrivate*: boolean value, true if the class is affected by *Class Data Should Be Private* smell, false otherwise;
    * *complexClass*: boolean value, true if the class is affected by *Complex Class* smell, false otherwise;

    * *functionalDecomposition*: boolean value, true if the class is affected by *Functional Decomposition* smell, false otherwise;
    * *godClass*: boolean value, true if the class is affected by *God Class* smell, false otherwise;
    * *spaghettiCode*: boolean value, true if the class is affected by *Spaghetti Code* smell, false otherwise;
    * *hasLongMethods*: boolean value, true if the class is affected by *Has Long Methods* smell, false otherwise;
    * *projectId*: reference key to the project;
    * *projectName*: project name;
    * *releaseEra*: chronological order of project's release inside the analysis scope. The oldest release has value 1, youngest one has the higher value.
  - View *commited_files_view* has been built by traversing the database as shown in Figure 4.6 and contains for each release the list of all modified files with the associated author. The actual code is shown in Algoritm 4.2. View structure is the following:
    * *authorId*: reference key to the commit author;
    * *releaseRangeId*: reference key to the exact moment in time;
    * *file*: path of the class's source file inside the project repository.
  - View *tech_and_community_smells_view*, with the exception of column *id*, contains the same columns with the same meaning of table *tech_and_community_smells*. Is possible to say that the "hard work" is done by this view and table *tech_and_community_smells* is just a place to store data for performance reasons. The actual code is shown in Algoritm 4.1.

- **Code and Community Smells association algorithm**. This series of steps aims to prepare the dataset that contains all the associations between Code and Community Smells, we will reason on that dataset in Chapter 5 when we will discuss about result evaluation of this empirical research. We have decided to consider only the code that is on master/default branch because, as industries practise suggests [7, 8], final (production) version of the code is located on that branch. Code and Community Smells association algorithm execution flow is the following:
  - Get list of releases from Codeface database. The Codeface release internal representation identify each of them with both a start and an end commit, we have decided to consider just the end commit as temporal significant moment in time; the risk of information loss consequent to the decision

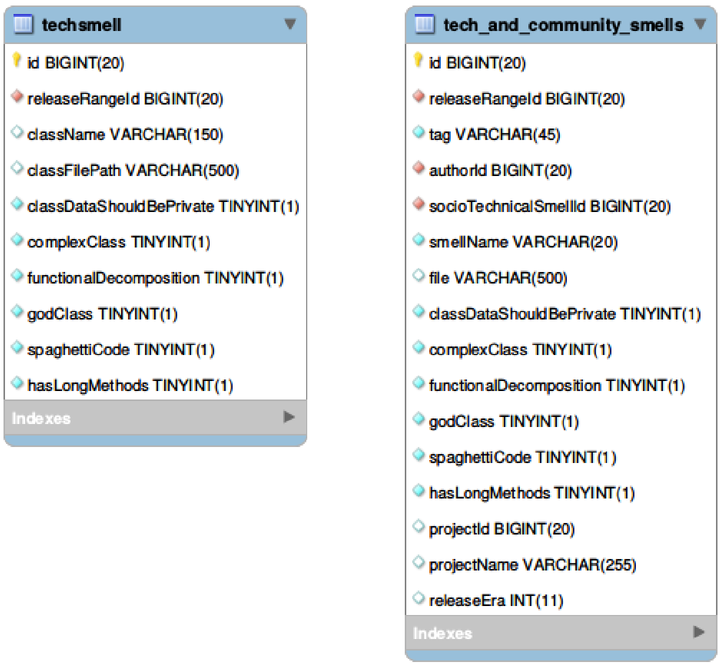


Figure 4.5: Technical analysis tables

we made is related to the very first release only, but, in our opinion, that risk is almost zero because of our goal as described in research question *RQ1a*;

– For each release we execute the following steps:

  1. Use Git *checkout* command to extract from the repository history the code that was in place in the exact moment when the release was created;
  2. Run *RunCodeSmellDetection.jar* to extract from the release code the complete list of all Code Smells which the release was affected by.

– Restore git *working directory* as it was at the beginning in order to, if needed, reproduce the analysis from the same initial conditions;

– Save csv reports in Codeface4Smells database table *techsmell*;

– Combine data from both Community and Technical Smells sources in order to produce list of occurrences and save it in table *tech_and_community_smells*. Data are combined using *tech_and_community_smells_view* SQL view directly inside Codeface4Smells database. The SQL view basically connects the some tables and views with idea of connecting files affected by at least one Code Smell to Community Smells using the "guilty" developer as common bridge between the two worlds. The actual code is shown in Algoritm 4.1;

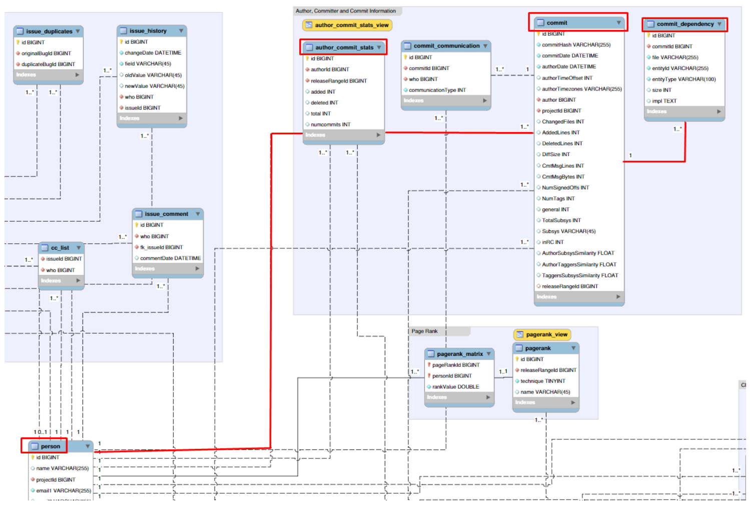


Figure 4.6: Technical analysis views

- Export project results from table *tech_and_community_smells* in csv format into project results directory.

To summarise, the value introduced with these contributions are:

- Is now possible to execute a combined Social and Technical analysis thanks to “tsa” command;
- The “tsa” command is fully integrated into the “smells” command;
- Data model has been enhanced and now new data are available for everyone.

4.3.4 Other enhancement

Other enhancement have been made on Codeface4Smells and its architecture which are indeed not in the exact scope of the experimental goal of this master thesis but still they made it possible. In this Section we are going to briefly summarise all of them.

- **Update Codeface code base**. Since Codeface4Smells is, at its extreme ratio, an extension of Codeface, we have started our journey with a modernisation of the code that both Codeface and Codeface4Smells have in common. We made this task in a manual fashion without the use of any automated tool in order to ensure the maximum level of carefulness during the whole task execution. During the execution of this task we have updated very critical part of Codeface starting from repository mining algorithm, to mailing list analysis up to data model extensions and modifications. The complete list of all changes made to the original version of the code is available online in the Codeface4Smells public repository [9].

Algorithm 4.1 Combination of Community and Code Smells

```
select   commited_files_view.releaseRangeId,
  release_timeline.tag,
  commited_files_view.authorId,
  sociotechnical_granular.socioTechnicalSmellId,
  sociotechnical_smells.name as smellName,
  commited_files_view.file ,
  techsmell.classDataShouldBePrivate,
  techsmell.complexClass,
  techsmell.functionalDecomposition,
  techsmell.godClass,
  techsmell.spaghettiCode,
  techsmell.hasLongMethods
from      commited_files_view
inner join sociotechnical_granular on commited_files_view.releaseRangeId =
    sociotechnical_granular.releaseRangeId and commited_files_view.authorId =
    sociotechnical_granular.personId
inner join techsmell on techsmell.releaseRangeId = commited_files_view.
    releaseRangeId and commited_files_view.file = techsmell.classFilePath
inner join release_timeline on commited_files_view.releaseRangeId = release_
    timeline.id
inner join sociotechnical_smells on sociotechnical_granular.socioTechnicalSmellId
    = sociotechnical_smells.id
where techsmell.classDataShouldBePrivate = 1
      or techsmell.complexClass = 1
      or techsmell.functionalDecomposition = 1
      or techsmell.godClass = 1
      or techsmell.spaghettiCode = 1
      or techsmell.hasLongMethods = 1;
```

Algorithm 4.2 List of Committed files per Release and Author

```
select distinct
  author_commit_stats.authorId,
  author_commit_stats.releaseRangeId,
  commit_dependency.file
from      author_commit_stats
inner join commit on author_commit_stats.releaseRangeId = commit.releaseRangeId
    and author_commit_stats.authorId = commit.author
inner join commit_dependency on commit.id = commit_dependency.commitId;
```

- **Fix social network analysis centrality metric computation**. In some edge case the computation of a centrality metric called Edgelist centrality was failing because of some wrong data type hypothesis made in code, we have fixed it by simply adding a safety check before the actual metric is computed. The original version is available in its online repository at [10] while the modified version is included in file "*$project-root-folder$/codeface/R/ml/ml_utils.r*"

To summarise, the values introduced with these contributions are:

- The Codeface code base has been updated with new features and some bug have been fixed;
- We have fixed a computation issue that was affecting an external library.

Chapter 5

Evaluation

This chapter presents the evaluation of obtained results inherent to the correlation between Community and Code Smells.

Considering Apriori algorithm [16] analysis results on the entire set of all analysed projects, it was possible to empirically identify correlations between Code and Community Smells. Within our analysis we considered a correlation as relevant if and only if its associated *support* was greater equal than 0.2 and its *level of confidence* was greater equal than 0.3, in order to determine which Community Smell could be considered as associated to one or more Code Smells.

Section 5.1 presents the evaluation of our results and addresses our Research Question *RQ1* and all its sub-questions, Section 5.2 specifies potential threats that could affect the validity of results, evaluations and conclusions of this study.

5.1 Correlations between Community and Code Smells

Considering the dataset selected from all the projects present in Table 3.1 we have been able to find a total of 9355 Community Smells occurrences summarised as in Table 5.1 and a total of 19507 Code Smells occurrences summarised as in Table 5.2.

Community Smell	Occurrences
Missing Links	5540
Radio Silence	2484
Organisational Silo	1331
ALL	9355

Table 5.1: Number of Community Smells Occurrences in the analysed dataset

In order to answer to *RQ1*, *RQ1a* and *RQ1b*, we have used Weka [38] implementation of Apriori algorithm, this algorithm is able to automatically identify all correlations between two or more factors. In order to decide the level of signifi-

Code Smell	Occurrences
Class Data Should Be Private	1794
Complex Class	3036
Functional Decomposition	180
God Class	7266
Spaghetti Code	5690
Has Long Methods	1541
ALL	19507

Table 5.2: Number of Code Smells Occurrences in the analysed dataset

cance of all the instances discovered by Weka we have set the minimum level of two quality factors. Support, indication of how frequently the occurrence appears in the dataset, was set to 0.2, as a consequence we are going to say that if this correlation happens less than once every five is just a coincidence. Confidence, indication of how often the rule has been found to be true, was set to 0.3, as a consequence we are going to say that, given the number of Community Smells occurrences, once the correlation happens if it happen for less than one every three Community Smells, is just a coincidence.

Apriori association rules must be interpreted as follows, the “head” implies the “body”. Starting from the previous assumption we can use Apriori results to understand at which level the existence of a Community Smell implies the existence of a Code Smell.

From the results in Table 5.3 we can see that both *Missing Links* and *Radio Silence* smells are correlated with *God Class* smell. Following Community Smells definitions from [70] and Code Smells definitions from [57] quite logically comes the conclusion that an absence in communication brings to monster classes that have the total control of big part of the software simply because if two developers are not aware of the existence of a specific functionality they will probably re-implement their own version inside their monster class.

Results from Table 5.3 shows also a correlation between *Missing Links* smell and *Spaghetti Code* smell, the lack of communication results in a lack of architectural organisation, since a developer is alone in building a functionality he will tend to think and organise his code in a procedural way because when you are alone in solving a problem is quite difficult to forecast generic patterns (which is a normal task in Object Oriented Programming) starting from that specific problem.

Result in Table 5.4 shown all the correlation the were rejected because below one of the two quality barrier that we had set at the beginning of this experiment. Even if this evidence are not statistically significant we can use them to qualitatively support the previous findings. Table 5.4 reports that *Radio Silence* smell seems to be correlated to *Spaghetti Code* smell, moreover *Organisational Silo* smell seems to have a weak correlation with *God Class* smell.

Head (Community Smell)	**Body (Code Smell)**	**Head Occurrences**	**Body Occurrences**	**Support**	**Confidence**
Missing Links	God Class	5540	4279	0.46	0.77
Missing Links	Spaghetti Code	5540	3364	0.36	0.61
Missing Links	God Class, Spaghetti Code	5540	3268	0.35	0.59
Radio Silence	God Class	2484	1911	0.2	0.77
			Average	0.343	0.685
			Min	0.2	0.59
			Max	0.46	0.77

Table 5.3: Apriori accepted findings

Head (Community Smell)	**Body (Code Smell)**	**Head Occurrences**	**Body Occurrences**	**Support**	**Confidence**
Missing Links	Complex Class	5540	1782	0.19	0.32
Missing Links	Complex Class, God Class	5540	1744	0.19	0.31
Missing Links	Complex Class, Spaghetti Code	5540	1644	0.18	0.3
Missing Links	Complex Class, God Class, Spaghetti Code	5540	1644	0.18	0.3
Radio Silence	Spaghetti Code	2484	1076	0.12	0.81
Radio Silence	Spaghetti Code, God Class	2484	1488	0.16	0.6
Organisational Silo	God Class	1331	1446	0.15	0.58
			Average	0.167	0.460
			Min	0.12	0.3
			Max	0.19	0.81

Table 5.4: Apriori rejected findings

Previously described findings and numbers in both Table 5.3 and Table 5.4 gives us the feeling of how deep correlation between Code and Community Smells is; this correlation has even more sense in the context of this experiment if one think that since *Missing Links*, *Radio Silence* and *Organisational Silo* are all "communicational" lack smells we can unsurprisingly conclude that they, more or less, have to be correlated with the same Code Smells.

Considering the results discussed till now, we can conclude and answer to our research questions as follows:

- **RQ1.** *Does a correlation between Community and Code Smells exists?* Yes, but partially. We have found out that both Missing Links and Radio Silence are statistically correlated to both God Class and Spaghetti Code. Nothing can be said with respect to all others Community Smells;

- **RQ1a.** *How strong this correlation is?* Empirical results of Table 5.3 tells us that correlation between *Missing Links* smell and both *God Class* and *Spaghetti Code* smells is quite relevant while correlation between *Radio Silence* smell and *God Class* smell is a bit less relevant;

- **RQ1b.** *Does this correlation changes between different Smells?* Yes. Based on our findings we can say that Missing Links have a stronger correlation with the above mentioned Code Smells with respect to Radio Silence correlation with the same Code Smells;

- **RQ1c.** *Is it possible to discover fine grained association between a Community Smell and the developers involved?* Yes. Artefacts described in Section 4.3.2 are able to gives us a fine grained link between a Community Smell and the developers involved;

- **RQ1d.** *Is it possible to detect Code Smells in a time line fashion?* Yes. Artefacts described in Section 4.3.3 are able to perform such a form of Code Smells detection;
- **RQ1e.** *Is it possible to approach the entire problem in an automated way?* Yes. Artefacts described in Section 4.3.1 solves this problem.

5.2 Threats to validity

This section highlights potential threats that could affect the validity of the results, evaluations and conclusions proposed in this master thesis.

Construct Validity. Threats to construct validity are related to the relationships between theory and observations and, generally, this typology of threats is mainly constituted by imprecision in performed measurements. The main threat is related to the precision of underlying tools to correctly identify both Code and Community Smells, therefore the study can be affected by construct validity.

Internal Validity. Threats to internal validity are related to factors that could have influenced our results. The correlation algorithm use all files modified by a developer in a release, this fact is a potential validity threat because that developer could have introduced a Community Smell while working on just a slice of all the files modified in a release, moreover that slice could have been Code Smell free.

External Validity. Threats to external validity are related to the generalisation of obtained results. Codeface4Smells currently identifies five different Community Smells and six different Code Smell, but there exists other Code Smells not considered in the static code analysis phase that might act as more significant indicators of the correlation existence. Moreover, we conducted our analysis on a total of ten Java FLOSS development projects, this fact highly influence the generalisation level of our findings mainly because of language and community type restriction.

Chapter 6

Conclusions and future work

This master thesis elaborates, operationalises, validates and discusses the existence of correlations between Code and Community Smells.

The evaluation of our work, presented in Chapter 5 allowed us to conclude that Community and Code Smells are actually correlated. Since our empirical evaluation has shown that Community Smells related to communication lacks are indeed correlated to Code Smells typical of lone developers works, the biggest insight we can get is to never stop communicating, especially if you a community leader, in order to maintain both technical and social level of quality as high as possible. Therefore, while executing project performance analysis, it is important to consider even the social and organisational aspects besides the technical ones, in order to lower the barriers that can influence the success of the development community.

With the goal of further increasing the correlation capabilities of our tool, a possible extension of this work can be the introduction of both Code and Community new Smells identification. Furthermore, a possible future enhancement can be the addition of different Code Smells identification tools in order to enlarge the programming languages analysis range. Moreover, a possible evolution of this software engineering research can consists in the evaluation of time series analysis, with the purpose of identifying causal relation between Community Smells and Code Smells.

Other possible directions in which this tool can evolve is identification of eradication costs associated to Community Smells or the introduction of new social network communication tools analysis such as online community forums or instant messaging group chat.

Finally, we plan to merge the developed tool-support in the main Codeface distribution, potentially transforming Codeface into a full-fledged continuous software development community improvement platform.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] `https://en.wikipedia.org/wiki/Globalization`.

[2] `http://www.gnu.org/philosophy/free-sw.html`.

[3] `https://opensource.org/osd.html`.

[4] `http://www.gnu.org/philosophy/open-source-misses-the-point.html`.

[5] `http://www.gnu.org/philosophy/floss-and-foss.html`.

[6] `http://siemens.github.io/codeface`.

[7] `http://nvie.com/posts/a-successful-git-branching-model/`.

[8] `https://docs.microsoft.com/en-us/vsts/git/concepts/git-branching-guidance`.

[9] `https://github.com/maelstromdat/CodeFace4Smells`.

[10] `https://github.com/wolfgangmauerer/snatm`.

[11] `https://www.vagrantup.com`.

[12] `https://linuxcontainers.org/`.

[13] `https://www.virtualbox.org`.

[14] `https://www.cs.waikato.ac.nz/ml/weka/`.

[15] `https://github.com/giarfa/Codeface4Smells`.

[16] Rakesh Agarwal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, pages 487–499, 1994.

[17] Nicolli SR Alves, Leilane F Ribeiro, Vivyane Caires, Thiago S Mendes, and Rodrigo O Spínola. Towards an ontology of terms on technical debt. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 1–7. IEEE, 2014.

[18] Matej Artač, Tadej Borovšak, Elisabetta Di Nitto, Michele Guerriero, and Damian A. Tamburri. Model-driven continuous deployment for quality devops. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*, pages 40–41. ACM, 2016.

[19] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143. ACM, 2006.

[20] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *20th International Symposium on Software Reliability Engineering*, pages 109–119. IEEE, 2009.

[21] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, pages 322–333. ACM, 2014.

[22] Frederick P Brooks Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[23] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[24] Marcelo Cataldo, James D Herbsleb, and Kathleen M Carley. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 2–11. ACM, 2008.

[25] Marcelo Cataldo, Patrick A Wagstrom, James D Herbsleb, and Kathleen M Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 353–362. ACM, 2006.

[26] Juyun Cho. Globalization and global software development. *Issues in information systems*, 8(2):287–290, 2007.

[27] Jorge Colazo. Structural changes associated with the temporal dispersion of teams: Evidence from open source software projects. In *47th Hawaii International Conference on System Sciences*, pages 300–309. IEEE, 2014.

[28] Lyra J Colfer and Carliss Y Baldwin. The mirroring hypothesis: Theory, evidence and exceptions. *Harvard Business School Finance Working Paper*, (16-124), 2016.

[29] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[30] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Assessing the bus factor of git repositories. In *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 499–503. IEEE, 2015.

[31] Ward Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992.

[32] Ivaldir H de Farias Junior, Ryan R de Azevedo, Hermano P de Moura, and Dennis S Martins da Silva. Elicitation of communication inherent risks in distributed software development. In *IEEE Seventh International Conference on Global Software Engineering Workshops*, pages 37–42. IEEE, 2012.

[33] Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, January 2017.

[34] Christopher P Earley and Elaine Mosakowski. Creating hybrid team cultures: An empirical test of transnational team functioning. *Academy of Management Journal*, 43(1):26–49, 2000.

[35] Neil A Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L Nord, and Ian Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 50–60. ACM, 2015.

[36] Martin Fowler. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

[37] Kenneth R Gray and Thomas L Friedman. The world is flat: A brief history of the twenty-first century, 2005.

[38] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[39] James D Herbsleb and Rebecca E Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE software*, 16(5):63, 1999.

[40] Qiaona Hong, Sunghun Kim, Shing Chi Cheung, and Christian Bird. Understanding a developer social network and its evolution. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 323–332. IEEE, 2011.

[41] James Howison, Keisuke Inoue, and Kevin Crowston. Social dynamics of free and open source team communications. In *IFIP International Conference on Open Source Systems*, pages 319–330. Springer, 2006.

[42] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. Mining and visualizing developer networks from version control systems. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 24–31. ACM, 2011.

[43] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. Classifying developers into core and peripheral: An empirical study on count and network metrics. *arXiv preprint arXiv:1604.00830*, 2016.

[44] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. From developer networks to verified communities: a fine-grained approach. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 563–573. IEEE Press, 2015.

[45] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6), 2012.

[46] 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 of the 37th International Conference on Software Engineering-Volume 1*, pages 677–687. IEEE Press, 2015.

[47] Fu-ren Lin and Chun-hung Chen. Developing and evaluating the social network analysis system for virtual teams in cyber communities. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 8–pp. IEEE, 2004.

[48] Luis Lopez-Fernandez, Gregorio Robles, Jesus M Gonzalez-Barahona, et al. Applying social network analysis to the information in cvs repositories. In *International workshop on mining software repositories*, pages 101–105, 2004.

[49] Gregory Madey, Vincent Freeh, and Renee Tynan. The open source software development phenomenon: An analysis based on social network theory. *AMCIS 2002 Proceedings*, page 247, 2002.

[50] Simone Magnoni. An approach to measure community smells in software development communities. mathesis, Politecnico di Milano, 2016.

[51] Simone Magnoni, Damian A. Tamburri, Elisabetta Di Nitto, and Rick Kazman. Discovering community smells: A quality model for software development communities. –.

[52] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[53] Andrew Meneely and Laurie Williams. Socio-technical developer networks: should we trust our measurements? In *Proceedings of the 33rd International Conference on Software Engineering*, pages 281–290. ACM, 2011.

[54] Eric Molleman and Jannes Slomp. The impact of team and work characteristics on team functioning. *Human Factors and Ergonomics in Manufacturing & Service Industries*, 16(1):1–15, 2006.

[55] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, pages 521–530. ACM, 2008.

[56] Ning Nan and Sanjeev Kumar. Joint effect of team structure and software architecture in open source software development. *IEEE Transactions on Engineering Management*, 60(3):592–603, 2013.

[57] Moha Naouel, Guéhéneuc Yann-Gaël, Duchien Laurence, and Le Meur Anne-Françoise. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, August 2009.

[58] Roozbeh Nia, Christian Bird, Premkumar Devanbu, and Vladimir Filkov. Validity of network analyses in open source projects. In *7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 201–209. IEEE, 2010.

[59] Fabio Palomba. *Code smells: relevance of the problem and novel detection techniques*. PhD thesis, Universita degli studi di Salerno, 2017.

[60] 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, 2015.

[61] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[62] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In *ICSME*, pages 91–100, 2014.

[63] Dirk Riehle, Philipp Riemer, Carsten Kolassa, and Michael Schmidt. Paid vs. volunteer work in open source. In *47th Hawaii International Conference on System Sciences*, pages 3286–3295. IEEE, 2014.

[64] Anita Sarma, Jim Herbsleb, and André Van Der Hoek. Challenges in measuring, understanding, and achieving social-technical congruence. In *Proceedings of Socio-Technical Congruence Workshop, In Conjuction With the International Conference on Software Engineering*, 2008.

[65] Seiji Sato, Hironori Washizaki, Yoshiaki Fukazawa, Sakae Inoue, Hiroyuki Ono, Yoshiiku Hanai, and Mikihiko Yamamoto. Effects of organizational changes on product metrics and defects. In *20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 132–139. IEEE, 2013.

[66] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.

[67] Damian A Tamburri and Elisabetta Di Nitto. When software architecture leads to social debt. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 61–64. IEEE, 2015.

[68] Damian A Tamburri, Philippe Kruchten, Patricia Lago, and Hans van Vliet. What is social debt in software engineering? In *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*, pages 93–96. IEEE, 2013.

[69] Damian A Tamburri, Patricia Lago, and Hans van Vliet. Organizational social structures for software engineering. *ACM Computing Surveys (CSUR)*, 46(1):3, 2013.

[70] Damian Andrew Tamburri, Philippe Kruchten, Patricia Lago, and Hans van Vliet. Social debt in software engineering: insights from industry. *J. Internet Services and Applications*, 6(1):10:1–10:17, 2015.

[71] Giuseppe Valetto, Mary Helander, Kate Ehrlich, Sunita Chulani, Mark Wegman, and Clay Williams. Using software repositories to investigate socio-technical congruence in development projects. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 25. IEEE Computer Society, 2007.

[72] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark GJ van den Brand, Alexander Serebrenik, Premkumar Devanbu, and Vladimir Filkov. Gender and tenure diversity in github teams. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3789–3798. ACM, 2015.

[73] Bogdan Vasilescu, Alexander Serebrenik, and Vladimir Filkov. A data set for social diversity studies of github teams. In *IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 514–517. IEEE, 2015.

[74] Christopher Vendome, Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk. When and why developers adopt and change software licenses. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 31–40. IEEE, 2015.

[75] Sunny Wong, Yuanfang Cai, Giuseppe Valetto, Georgi Simeonov, and Kanwarpreet Sethi. Design rule hierarchies and parallelism in software development tasks. In *24th IEEE/ACM International Conference on Automated Software Engineering*, pages 197–208. IEEE/ACM, November 2009.

[76] Nianjun Zhou, Wesley M Gifford, Krishna Ratakonda, Gregory H Westerwick, and Carl Engel. On the quantification of global team performance and profitability. In *Services Computing (SCC), 2014 IEEE International Conference on*, pages 378–385. IEEE, 2014.

Appendix A

Codeface4Smells

This Appendix describes how to install, set-up and execute Codeface4Smells, the tool developed during this master thesis. Moreover we report a set of notions on how to manually write a proper configuration file useful in case of manual analysis or for advanced analysis customisation.

A.1 Set-up and analysis execution

Codeface4Smells is hosted on GitHub and then it is possible to download it manually or cloning its associated git repository [9]. As explained in Chapter 4, Codeface4Smells extends Codeface and thus, it shares its inner workings. Therefore, since Codeface is supposed to be executed in a virtual machine, for practical reasons, the presence of Vagrant [11] and, either Linux LXC [12] or VirtualBox [13] on the machine is a necessary precondition in order to continue with the set-up of our tool.

Once the source code of Codeface4Smells is present on the machine, it is possible to initialise the virtual machine and then access it, using the following set of commands:

```
# vagrant up
# vagrant ssh
```

The first command initialises the virtual machine. If it is the first time that the considered instance of Codeface4Smells is executed, then Vagrant will download the virtual machine image and some initialisation scripts will download and install all the necessary software to correctly execute the tool. The second command establishes an ssh connection with the previously initialised virtual machine, in order to have access to its console.

Whenever the virtual machine is not needed anymore, it is possible to shut it down using the following command:

```
# vagrant halt
```

Once you are connected to the virtual machine console, all you need to do before starting a new analysis job is to create the folder of the project you want to analyse. After the folder creation you have to navigate inside the folder and then execute Codefa4Smells analysis. The complete list of all available projects is present in Table 3.1. Starting from now, thanks to all contributions described in Section 4.3 and followings, the software will take care of everything. Therefore, the list of commands necessary to correctly execute Codeface4Smells analysis is:

```
# mkdir <PROJECT_NAME>
# cd <PROJECT_NAME>
# codeface smells
```

A.2 Project configuration

Even if, thanks to automated analysis described is Section 4.3.1, is not mandatory to manually compile an analysis configuration file, can be useful to see details of how to manually compile one in order to have a deeper control in case of manual analysis. To perform an analysis with Codeface4Smells, it is necessary to specify a configuration file that contains all the necessary information related to the project in analysis, it must have a ".conf" extension and it should contain at least the following parameters:

- *project*: name of the project to analyse;
- *repo*: name of the directory containing the source code;
- *mailinglists*: lists of mailing lists names and typologies;
- *description*: description of the project to analyse;
- *revisions*: lists of versions to analyse. If this parameter is not present all the project history will be analysed using three months windows. It is possible to set this parameter to "3months" in order to analyse at most the last three years of activity using three months windows;
- *tagging*: collaboration detection method.

An example of a valid configuration file, which allows to analyse Cassandra project, is the following:

```
project: Cassandra
repo: cassandra
mailinglists:
        - {name: gmane.comp.db.cassandra.devel,
           type: dev, source: gmane}
```

```
description: Cassandra project
revisions: 3months
tagging: proximity
```

Appendix B

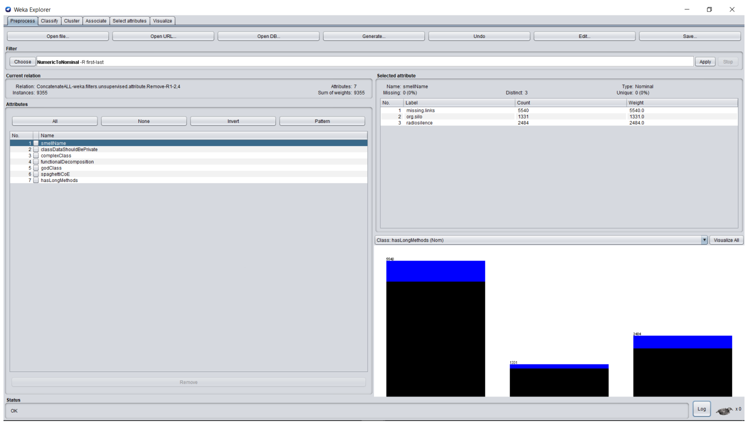
Empirical results replication

This appendix contains the list of all steps necessary to replicate the dataset analysed and evaluated in Chapter 5.

In order to use Weka implementation of Apriori algorithm as we did in Section 5.1 the presence of Weka [14] is a necessary precondition.

Once you have Weka installed on your machine you can download and open in *Weka Explorer* the pre-compiled dataset csv file available at [15].

Figure B.1 shows how to setup *Attributes* in *Weka Explorer Preprocess* tab, while Figure B.2 describes how to setup Apriori parameters *Weka Explorer Associate* tab.



Filter: NumericToNominal -R first-last

Current relation
Relation: ConcatenateALL-weka.filters.unsupervised.attribute.Remove-R1-2,4
Instances: 9355
Attributes: 7
Sum of weights: 9355

Attributes

No.	Name
1	smellName
2	classDataShouldBePrivate
3	complexClass
4	functionalDecomposition
5	godClass
6	spaghettiCoE
7	hasLongMethods

Selected attribute
Name: smellName
Type: Nominal
Missing: 0 (0%)
Distinct: 3
Unique: 0 (0%)

No.	Label	Count	Weight
1	missing.links	5540	5540.0
2	org.silo	1331	1331.0
3	radiosilence	2484	2484.0

Class: hasLongMethods (Nom)

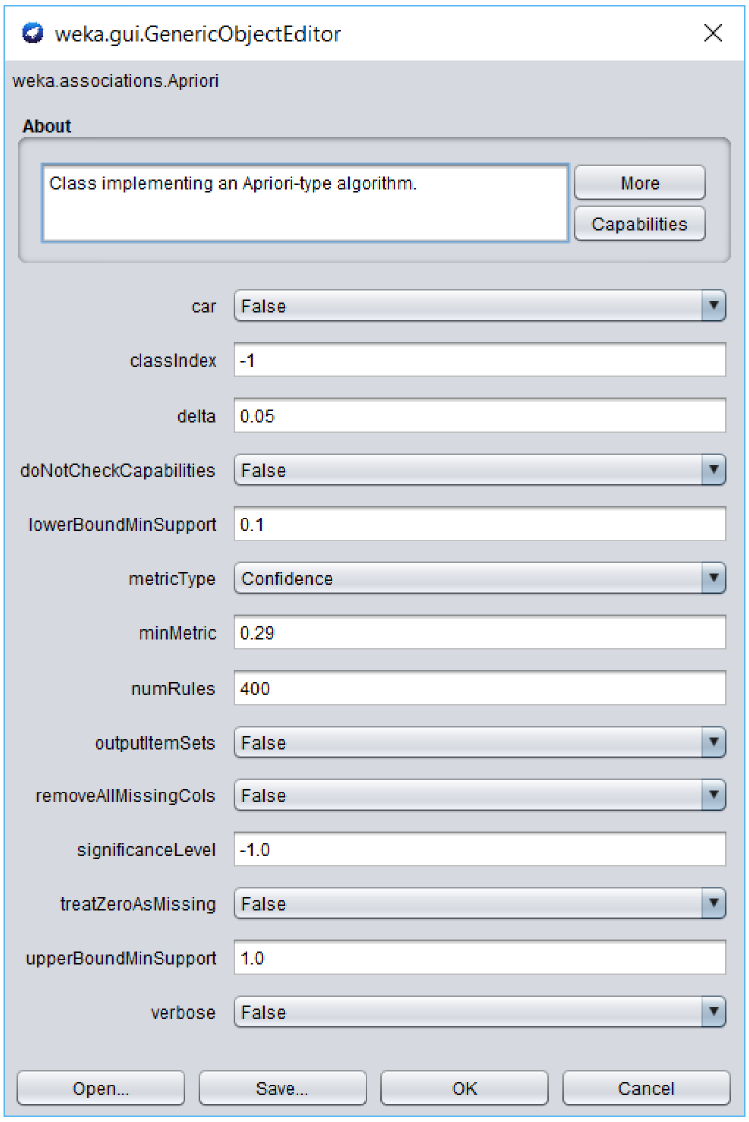
Status: OK

Figure B.1: Weka Preprocess Attributes configuration

Once everything is properly configured you should see the following string in the *Associator* box:

```
Apriori -N 400 -T 0 -C 0.29 -D 0.05 -U 1.0 -M 0.1 -S -1.0 -c -1
```

You can now start a new Apriori evaluation which will produce the same results described in Section 5.1.


weka.gui.GenericObjectEditor

weka.associations.Apriori

About

Class implementing an Apriori-type algorithm.

car	False
classIndex	-1
delta	0.05
doNotCheckCapabilities	False
lowerBoundMinSupport	0.1
metricType	Confidence
minMetric	0.29
numRules	400
outputItemSets	False
removeAllMissingCols	False
significanceLevel	-1.0
treatZeroAsMissing	False
upperBoundMinSupport	1.0
verbose	False

Figure B.2: Weka Apriori parameters configuration