UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Structural and Evolutionary Analysis of Developer Networks

Mitchell Joblin

February 18, 2017

Dissertation zur Erlangung des Doktorgrades
der Naturwissenschaften (Dr. rer. nat.)
eingereicht an der Fakultät für Informatik und Mathematik
der Universität Passau

Dissertation submitted to
the Department of Informatics and Mathematics of
the University of Passau
in Partial Fulfillment of Obtaining
the Degree of a Doctor in the Domain of Science

Betreuer / Advisors:
Prof. Dr. Sven Apel
Prof. Dr. Wolfgang Mauerer

Externer Gutachter / External Examiner:
Prof. Dr. Massimiliano Di Penta

Abstract

Large-scale software engineering projects are often distributed among a number sites that are geographically separated by a substantial distance. In globally distributed software projects, time zone issues, language and cultural barriers, and a lack of familiarity among members of different sites all introduce coordination complexity and present significant obstacles to achieving a coordinated effort.

For large-scale software engineering projects to satisfy their scheduling and quality goals, many developers must be capable of completing work items in parallel. A key factor to achieving this goal is to remove interdependencies among work items insofar as possible. By applying principles of modularity, work item interdependence can be reduced, but not removed entirely. As a result of uncertainty during the design and implementation phases and incomplete or misunderstood design intents, dependencies between work items inevitably arises and leads to requirements for developers to coordinate. The capacity of a project to satisfy coordination needs depends on how the work items are distributed among developers and how developers are organizationally arranged, among other factors. When coordination requirements fail to be recognized and appropriately managed, anecdotal evidence and prior empirical studies indicate that this condition results in decreased product quality and developer productivity. In essence, properties of the socio-technical environment, comprised of developers and the tasks they must complete, provides important insights concerning the project's capacity to meet product quality and scheduling goals.

In this dissertation, we make contributions to support socio-technical analyses of software projects by developing approaches for abstracting and analyzing the technical and social activities of developers. More specifically, we propose a fine-grained, verifiable, and fully automated approach to obtain a proper view on developer coordination, based on commit information and source-code structure, mined from version-control systems. We apply methodology from network analysis and machine learning to identify developer communities automatically. To evaluate our approach, we analyze ten open-source projects with complex and active histories, written in various programming languages. By surveying 53 open-source developers from the ten projects, we validate the accuracy of the extracted developer network and the authenticity of the inferred community structure. Our results indicate that developers of open-source projects form statistically significant community structures and this particular network view largely coincides with developers' perceptions.

Equipped with a valid network view on developer coordination, we extend our approach to analyze the evolutionary nature of developer coordination. By means of a longitudinal empirical study of 18 large open-source projects, we examine and discuss the evolutionary principles that govern the coordination of developers. We found that the implicit and self-organizing structure of developer coordination is ubiquitously described by non-random organizational principles that defy conventional software-engineering wisdom. In particular, we found that: (a) developers form scale-free networks, in which the majority of coordination requirements arise among an extremely small number of developers, (b) developers tend to accumulate coordination requirements with more and more developers over time, presumably limited by an upper bound, and (c) initially developers are hierarchically arranged, but over time, form a hybrid structure, in which highly central developers are hierarchically arranged and all other developers are not. Our results suggest that the organizational structure of large software projects is constrained to evolve towards a state that balances the costs and benefits of coordination, and the mechanisms used to achieve this state depend on the project's scale.

As a final contribution, we use developer networks to establish a richer understanding of the different roles that developers play in a project. Developers of open-source projects are often classified according to core and peripheral roles. Typically, count-based operationalizations, which rely on simple counts of individual developer activities (e.g., number of commits), are used for this purpose, but there is concern regarding their validity and ability to elicit meaningful insights. To shed light on this issue, we investigate whether count-based operationalizations of developer roles produce consistent results, and we validate them with respect to developers' perceptions by surveying 166 developers. We improve over the state of the art by proposing a relational perspective on developer roles, using our fine-grained developer networks, and by examining developer roles in terms of developers' positions and stability within the developer network. In a study of 10 substantial open-source projects, we found that the primary difference between the count-based and our proposed network-based core–peripheral operationalizations is that the network-based ones agree more with developer perception than count-based ones. Furthermore, we demonstrate that a relational perspective can reveal further meaningful insights, such as that core developers exhibit high positional stability, upper positions in the hierarchy, and high levels of coordination with other core developers, which confirms assumptions of previous work.

Overall, our research demonstrates that data stored in software repositories, paired with appropriate analysis approaches, can elicit valuable, practical, and valid insights concerning socio-technical aspects of software development.

Acknowledgements

A doctoral dissertation cannot be successfully completed in isolation. Instead, one must rely on the earlier efforts of individuals that provided the solid epistemic foundation on which future work could build upon. I am grateful to these countless individuals for their formidable efforts and I hope that my contributions to the field provide a useful basis for future research. During my time as a student, I was also extremely fortunate to have the direct support of incredibly talented and inspiring people.

First and foremost, I would like to express gratitude to my advisor Sven Apel. A great idea often begins in a fragile state and can be swiftly crushed by only a few words. Sven's ability to foster an idea, patiently develop it and deliver criticism with a degree of sensitivity, is a truly valuable and unique quality. I have incredible respect for his enthusiasm for doing scientific work and seemingly limitless dedication to supporting students. It is with this attitude that a positive environment where research can thrive is made possible. His strong appreciation for nuance has had a profound influence on my development as a person and as a researcher. It has been an absolute privilege to work with Sven.

I would also like to thank my second advisor Wolfgang Mauerer. In the ideal case, software engineering research should be aimed at providing a benefit to practitioners. For this reason, I am extremely grateful for all of Wolfgang's helpful comments and guidance coming from an industry perspective. The involvement of an industry partner has undoubtedly made the outcome of my work valuable to a broader audience.

I would also like to thank my external examiner Massimiliano Di Penta. Evaluating a doctoral thesis is an arduous task, and I am very appreciative for your willingness to accept this duty.

I want to thank the research group at the University of Passau. It was always such a pleasure for me to visit. Thank you to Norbert Siegmund, Janet Siegmund, Claus Hunsen, and Olaf Lessenich for the helpful comments and involvement with the empirical studies.

Lastly, I want to thank my friends and family. I am especially thankful to my parents for supporting me in all my pursuits and tolerating me as a child, who at times, experimented a little too often with testing the boundaries. It is with your enduring patience that my passion for experimentation and exploration was not crushed at childhood. To my partner Dianna, your relentless pursuit of perfection, work ethics, and courage in the face of an equation has been an inspiration to me. The countless hours of thought provoking discussions on my research and yours has certainly made me question our sanity at times, but also improved the work considerably. Thank you.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Problem and Motivation

Software is playing an increasingly central role in society. In part, this is due to the fact that software is no longer restricted to controlling primitive systems with a highly limited ability to influence the real world. Critical infrastructure (nuclear reactors, power transmission, telecommunications), transportation vehicles (airplanes, trains, automobiles, sea vessels), and medical equipment (imaging modalities, radiotherapy treatment, surgical robotics) are just a few examples of domains that pose a substantial threat to public safety and the global economy when software fails to behave according to expectations [Kni02; SCC$^+$12]. As more and more industries aggressively invest in software solutions, it is imperative that software engineering matures into a discipline that can efficiently and reliably generate high quality assets [And11]. Although conjecture of a "software gap"—which posits the existence of a widening gap between ambitions and achievements in software engineering—has been discussed as far back as the 1960s [NR69], the severe risks associated with software-intensive systems are still present today. Significant achievements have been made since the 1960s in terms of tools and techniques to support efficient production of high quality software, however, progress still appears to be outpaced by ambitions. To this day, it is not uncommon for software projects to experience substantial delays, deployment of severely inadequate systems, and high failure rates, which are indicative of a discipline in its infancy [ES13; EK08; Mar; SCC$^+$12].

Recently, in an effort to comply with aggressive project deadlines and budgets, yet still satisfy incredibly ambitious software requirements intrinsic to large and complex systems, software engineering challenges have changed considerably. In particular, the changes have had a substantial impact in terms of human factors and their influence on project success. A key trend occurring in software engineering is the transition from small collocated projects towards large-scale globally-distributed projects. Time zone issues, language and cultural barriers, and unfamiliarity with co-workers, all present formidable challenges to achieving project success in large-scale multi-site projects [ESKH07; HM03a; HMFG00]. To overcome these challenges, human factors must be considered in the context of technical factors, not simply viewed in isolation or ignored all together. On this basis, many researchers, including us, find it absolutely necessary to adopt a socio-technical view point on software projects [BMB$^+$07; BNM$^+$11; CH13; dSR11; ESKH07; MWSO08]. Specifically, developer coordination—the management of dependencies among tasks [MC90]—is substantially more difficult in a globally distributed work environment. Work items must be split, assigned to individuals or teams, independently completed in parallel, and then coherently assembled into a functionally complete system. Depending on how work items are divided, assigned, and temporally ordered, the complexity of coordinating the efforts of developers can vary substantially [CH13]. In two recent high-profile case studies, the deployment of HealthCare.gov[1] and NASA's Mars Climate Orbiter, factors related to inadequate coordination were cited as primary causes in the failure investigation reports. Specific statements regarding the causes of failure for HealthCare.gov include "…lack of an overarching project leader complicated the system's development because contractors received absolutely conflicting direction between the various entities within CMS [Centers for Medicare & Medicaid Services]" [ES13]. In the case of the Mars Climate Orbiter "Inadequate communications between project elements" was cited as one of the eight main causes of the disaster [Mar].

Software engineering researchers have made notable progress in terms of the development of tools and techniques for managing coordination complexity. Issue and bug trackers (Jira[2], Bugzilla[3]), version-control systems (Subversion[4],

[1]The Obama administration dedicated one of their principal initiatives to making healthcare easily and affordably available to every citizen. HealthCare.gov was the Web site which served as the primary interface to this initiative. Serious technical issues lead to a situation where thousands of people thought they had healthcare insurance, when in reality, they did not. The initial budget for the Web site's development was $93.7 million and grew to $1.7 billion at completion.

[2]`www.atlassian.com/software/jira/`

[3]`www.bugzilla.org/`

[4]`www.subversion.apache.org/`

Git[5]), software process models (waterfall model, agile practices), software project management tools (launchpad[6]), private e-mail messaging and instant messaging services, public mailing lists, virtual meetings (Skype), social coding platforms (Github[7]), wikis and product structure (modularity, information hiding, component-based software engineering), can all be viewed in one way or another as approaches for supporting coordination among software engineers. Interestingly, some projects seem to benefit more than others from a given approach, what works in one case, fails miserably in another [HPB05]. The myriad of available options is a testament to the importance of supporting coordination in software engineering, however, we still lack a fundamental understanding of how each approach impacts developer coordination and which specific coordination problem is appropriately solved by which specific solution. Essentially, the primary means of recourse, that is currently at our disposal, is to exhaustively search the densely populated landscape of tools and techniques until one is found that seems to produce the intended outcome. This search process is obviously resource intensive and often reaches only suboptimal solutions in the end, if at all.

Problems of coordination arise in many situations and we can draw inspiration from more mature fields providing hints for how we can establish foundational knowledge regarding coordination among people in software engineering projects. A related area that is substantially more well developed and understood is parallel computing, which requires solutions to similar problems we face in the coordination of people. For example, program parallelization requires an appropriate decomposition of a large problem into smaller sub-problems that can be quickly solved independently and concurrently, insofar as possible, such that the results of the sub-problems can be efficiently reconciled into a complete solution [Kum02]. Parallel program execution requires careful consideration of resource dependencies and specialized mechanisms and policies (e.g., scheduling, mutual exclusion, synchronization points) to control how parallel execution is able to proceed in order to avoid violating rules that would invalidate the result. The critical foundation required for realizing program parallelization is fundamental knowledge of dependencies between program elements. For example, a deep understanding of how control structures in the program and utilization of common resources (e.g., disk, memory) influence the complexity and overhead of parallel execution. It is with this fundamental knowledge that we are capable of recognizing when the overhead associated with parallelization outweigh the benefit of concurrent execution and how to

[5]`www.git-scm.com/`
[6]`www.launchpad.net/`
[7]`www.github.com/`

strategically and automatically optimize a sequence of operations to achieve a maximal benefit from parallelization [BEN$^+$93]. In the case of coordination among people, an immensely beneficial piece of information is knowing when adding developers to a project will not achieve a benefit in terms of faster development or higher product quality [Bro78]. In the developer coordination problem, we currently lack the fundamental understanding of the structure and evolution of coordination dependencies among developers to establish theories of coordination and coordination policies for software development. In this thesis, we strive to shed light on this area by adopting a first principles approach through reasoning from the basic elements of coordination, in terms of the pairwise activity of developers, up to the global structure that emerges from viewing those pairwise interactions as one collective system representing the overall coordination structure of a software project.

1.2 Thesis Goal

Overall, our goal is to enable quantitative analysis on socio-technical aspects of software development by leveraging data stored in commonly available software repositories. In particular, we address the question of how techniques and theory from network science (e.g., relational abstraction, social network analysis, random graph theory) can be applied to model and analyze properties of developer coordination. We develop, evaluate, and compare different approaches of abstracting the activities of developers into a network representation. We also explore and validate techniques for extracting structural and temporal patterns from complex networks that are a representation of the global coordination structure of a software project. We demonstrate how a network perspective can provide richer insights and outperform standard variable representations in certain tasks (e.g., classifying developers according to the role they fulfill). Our focus in this thesis is on developer coordination during the implementation and maintenance phases of the software life cycle. While there are certainly other phases where coordination is required, implementation and maintenance typically represent the longest and most expensive phases [BR00]. On this basis, we reason that improvement in coordination during these phases results in the largest benefit to a project in terms of achieving scheduling and budgeting goals.

1.3 Contributions

In this thesis, we make several contributions to facilitate the analysis and understanding of socio-technical aspects of software development. The extent of our contributions include (1) proposing and evaluating approaches for abstracting the activities of software developers into a network representation, known as *developer networks*, (2) identifying several organizational principles that manifest as structural and temporal patterns in developer networks, (3) validating insights drawn from developer networks in terms of both statistical and real-world significance. We employ empirical research methodology using both independent and direct data collection techniques [LSS05] to study the socio-technical structure of numerous high-profile software projects with wide-spread deployment (e.g., Linux Kernel, Mozilla Firefox, and Apache Web Server). We utilize independent data collection techniques by analyzing the output and by-products of software engineering work using software repository mining and static analysis approaches. We validate insights elicited from the analysis based on the independent techniques with data collected using the direct technique of Web-based personal opinion surveys of open-source developers. Specifically, we make the following three main contributions.

1. **Authentic Developer Network:** Our first contribution is to develop an approach of abstracting task-based interdependencies among developers as a complex network. We measure success based on the extent to which the network represents an authentic view on the developer organizational structure by exhibiting real-world validity. We compare the performance of several network construction approaches, all of which are based on data stored in the version-control system. In particular, the approaches that we evaluate make use of various data including: source-code files, manually reported references to developer participation, commit metadata, and a novel approach using source-code structure. We perform an empirical study on 10 substantial open-source projects and conduct a survey of 56 open-source developers drawn from these 10 subject projects. Our results indicate that developer networks constructed using our proposed fine-grained technique—which uses static source-code analysis to localize software changes to source-code entities—are an authentic reflection of most developers perception of the project's organization. An additional benefit of our fine-grained approach is that it allows us to infer the developer community structure using unsupervised machine learning techniques, which is not possible with other approaches. We found that the communities represent both a statistically significant organizational

pattern and reflect most developers' perception of which developer groups are involved in highly interdependent tasks.

2. **Network Structural & Temporal Patterns:** Our second contribution is to extend the developer network construction approach, which was conceived and validated in contribution 1, to identify evolutionary principles that govern the evolution of developer coordination. We adapt the network construction approach to include a time component by generating a sequence of developer networks that model the time-varying structure of developer coordination. We perform a longitudinal study of 18 substantial open-source projects, in some cases, consisting of more than 20 years and 500,000 commits of software development history. The results of our study indicate that the structural evolution of developer coordination changes over time according to several non-random organizational principles and that the socio-technical conditions in early project phases are fundamentally transformed as the project matures and grows. In particular, as a project grows, the coordination burden shifts towards severe inequality so that the majority of coordination is managed by an extremely small portion of developers. We found that this shift towards inequality coincides with projects that are capable of achieving sustainable super-linear growth in the number of contributing developers. We found that in the early phases of a project, developers are structurally arranged in a hierarchy, but as the project matures, this global hierarchy decomposes as the organization becomes more distributed. However, the highly active developers of a project appear to always remain hierarchically structured regardless of project phase. These patterns and insights provide direction as to which coordination challenges the project faces during particular project phases. Furthermore, the results suggest that depending on a developer's position in the organizational structure, the nature of the coordination challenges they face are substantially different. In essence, different tools and techniques are needed to support different positions in the organization and during different periods of time during the project's development.

3. **Developer Role Classification:** In our third contribution, we make use of the validated network construction approach (contribution 1) and the network's structural and temporal patterns (contribution 2) to automatically classify developers according to specific roles. Open-source developers fulfill different roles that each have distinctive qualities and purposes. For example, developers in a core role typically have long-term involvement, extremely well-developed technical knowledge and

strong leadership positions, while developers in a peripheral role typically contribute irregularly and their technical contributions are occasionally inadequate (e.g., violate design rules, introduce defects) [FPB+15; TRC10]. Still, successful open-source projects are ubiquitously comprised of a relatively small core developer group with a comparatively enormous peripheral group, and it appears that the interplay between these groups is critical to project success [JS07; Ray99]. Typically, developer roles are operationalized as simple one-dimensional metrics that count the activity of a developer (e.g., number of commits contributed). Unfortunately, these simple counts provide very limited insight into the relationship between developers occupying the same or different roles. As a contribution, we propose operationalizations based in relational abstraction in terms of the inter-developer relationships captured by our developer networks. We also include information on developer–developer communication by mining the projects' e-mail archives. More specifically, we utilize knowledge of a developer's position and stability within the organizational structure to identify their role. We performed an empirical study on 10 substantial open-source projects to compare the results generated by the different operationalizations. Our results indicate that core and peripheral developers are not only defined based on their levels of activity or engagement within a project, but they represent organizationally distinctive entities with substantially different coordination challenges. For example, we find that core developers are positionally stable and located at the top of the organizational hierarchy and that they constitute the most heavily coordinated group. In comparison, peripheral developers tend to be positionally unstable and are more likely to work on tasks that are coordinated by members of the core group rather than other peripheral developers. To evaluate which operationalizations most closely reflect developer perception, we conducted a Web-based survey of 166 open-source developers drawn from the 10 subject projects. We found that to a large extent developer perception substantiates all operationalizations. However, a network-based operationalization always outperforms the corresponding count-based operationalization in terms of more closely reflecting developer perception. Overall, the network-based operationalizations deliver more meaningful insights into the interplay between developer roles and holds greater real-world validity compared to non-relational operationalizations. From the survey we conducted, we found that developers often define roles in terms of modes of interaction with other developers. For example, one developer commented "core maintainers participate in discussions on areas outside the ones they maintain".

We think that the relational operationalizations perform better because they are able to capture the rich interaction patterns among developers, which provide meaningful insight that count-based metrics cannot.

1.4 Outline

In Chapter 2 (*Background*) we provide an overview of the background concepts to establish the foundational basis on which the research in this dissertation is built. In particular, we introduce the problem of coordination in software engineering and factors that cause an inevitable need for developers to coordinate their efforts. Next, we introduce software repositories as a means to elicit insight into the social and technical activities of developers. Network science plays an important role in our work and for this reason we provide an overview of both the tools and techniques of network analysis in addition to emphasizing the fundamentals of a network perspective and the nuances concerned with abstracting real-world phenomena as a network. We conclude the chapter with a introduction to the basics of constructing developer networks from software repository data. In Chapter 3 (*Community Detection and Validation with Fine-grained Developer Networks*), we introduce our novel approach for constructing developer networks, based on knowledge of source-code structure. In addition, we make comparisons with the the state-of-the-art approach by examining the networks' community structure in terms of both statistical and real-world significance. In Chapter 4 (*Evolutionary Trends of Developer Coordination*), we add in the time component to explore developer network dynamics that influence the evolution of developer coordination. We identify a number of evolutionary patterns that manifest in terms of fundamental transformations in the structural features of the network. In Chapter 5 (*Classifying Developers into Core and Peripheral Roles*), we make use of developer networks and the associated evolutionary patterns to perform the task of classifying developers into the role they fulfill. In this chapter, we emphasize the practical value of developer networks by demonstrating a situation in which a network perspective outperforms alternative representations. Finally, in Chapter 6 (*Conclusion and Future Work*), we discuss the broader implications of our work for analyzing and understanding socio-technical aspects of software engineering and provide opportunities for future work.

CHAPTER 2

Background

This chapter provides a concise introduction to the foundational concepts that support the research in this dissertation. We begin by examining coordination in software engineering with attention given to the mechanisms that cause an inevitable need for developers to coordinate their work. This is followed by the introduction of software repositories as a means to evaluate software projects and elicit meaningful insights that are relevant to the field of software engineering. Next, we discuss networks as a relational abstraction on real-world phenomena, in comparison to standard variable representations, and introduce the tools of complex network analysis. The chapter concludes with a discussion on network abstractions of the social and technical activities that developers engage in during software development.

2.1 Coordination in Software Engineering

In Watts Humphrey's seminal work on software quality, he concluded the following: "People are the organization's most important asset" [Hum96]. Humphrey's rationale was based on the principle that expecting the introduction of defects to software and then employing a reactive strategy to fix them is inefficient and costly. By changing the focus to people and how they performed their work, quality could be assured earlier in the software process resulting in cost and efficiency advantages. In the almost 20 years since Humphrey's realization, the importance of people and how they perform their work has only increased. As a response to severe pressure to reduce time-to-market and the desire to remain competitive by tapping into global talent pools, organizations are increasingly adopting global software development practices. These organizations now face more and more challenges stemming from the need to coordinate large numbers of individuals working on tasks that span geographic, cultural, and language boundaries [BMB$^+$07; HG99; HMFG00; ORM03].

Concurrent development The benefit of involving a large number of developers in the construction of a software system is achieved through concurrent work assignment. Typically, managers use high-level design documents to divide the development effort into task assignments that are independent enough to be completed in isolation from the vast majority of system elements [Con68; EWSG94; SGCH01]. The effectiveness of this divide-and-conquer strategy relies heavily on a sensible decomposition of the overall software system into elements that are minimally interdependent. The primary assumption here is that a modular design structure leads to a modular task structure. In theory, by reducing the interdependence between system elements, the likelihood of changes in one element causing unintentional side-effects in another elements, is reduced, so that task assignments can be safely carried out in parallel. In Parnas' highly influential work on modularity, he recognized that the criteria used to decompose a system affects the efficacy of concurrent development and that modules should be thought of as work assignments [Par72]. By using principles of information hiding, he demonstrated that independent development of software modules can be successfully realized. Despite this important insight, ideal system decompositions are most often stifled by over-simplified views on technical dependencies and diminishing similarity between task and product structure over time [BMB$^+$07; GGH$^+$07]. The result is that, in reality, many developer's tasks inevitably become interdependent in ways that are often unexpected and unavoidable [BMB$^+$07; KS95].

Coordination breakdowns At some point in time, independently developed modules must be composed to generate a functionally complete system. A primary cause of task interdependence is tasks that span one or more technical dependencies [HG99; KS95; ORM03]. An example of this is when one developer's task involves the consumption of data that is produced by a module implemented or modified by another developer. A depiction of this scenario is shown in Figure 2.1. In this example, the task interdependency stems from a data-flow dependency between modules, but similar issues can arise from a variety of dependency mechanisms (e.g., structural dependency, behavior dependency, evolutionary dependency, shared resource, state management, project scheduling) [BMB$^+$07; CMRH09]. When assumptions embedded in the

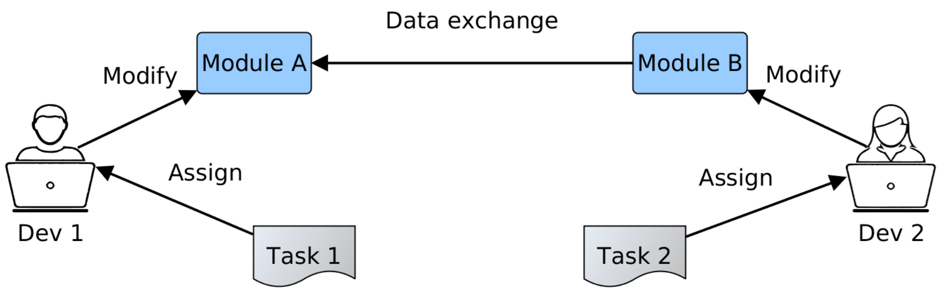


Figure 2.1: Two developers are assigned tasks that required modifications to modules A and B. There is a data dependency that exists between the modules that leads to a coordination requirement between the developers. Developer 1 is constrained by modifications performed by developer 2 that in any way affect the nature of data exchanged.

format or content of the exchanged data are not sufficiently understood by the developers involved, the resulting implementation is likely to contain defects. In a famous high profile example, a mishap investigation board determined that the Mars Climate Orbiter disaster was primarily a result of inadequate coordination between two geographically distributed teams [Mar]. In this case, the teams failed to adequately coordinate, which led to the misinterpretation of units of measurement that were exchanged between two independently developed modules. Consequently, a *coordination requirement* exists between the developers by virtue of the task interdependence. The coordination requirement represents a constraint or set of constraints that must be satisfied to successfully complete the corresponding tasks. Through implicit or explicit means, the developers must coordinate their efforts to avoid introducing defects. Explicit means of coordination occur when developers rely on direct interactions (e.g., face-to-face discussions, e-mails, phone calls) to exchange knowledge and resolve coordination requirements [KS95]. In some cases, the artifacts themselves (e.g., code comments or the code itself) or historical information provided by previous changes to the code provide sufficient information to satisfy a coordination requirement without resorting to explicit means of coordination [Hey07]. In large-scale, globally-distributed projects, it is not possible for a single person, or even a small group of people, to have detailed and complete knowledge of all coordination requirements between developers [CKI88; HMFG00; KS95]. This leads to project conditions where developers can easily lose awareness of the coordination requirements relevant to them or become overwhelmed by too many coordination requirements [dSQTR07]. There is a growing body of empirical evidence suggesting that complexity in coordination requirements, and the inability to adequately manage them, poses a significant threat to project success factors [CH13]. Cataldo et al. found that the number of coordination requirements centered on a file is more indicative of its failure proneness than the number of structural dependencies (e.g., data flow and function call) involving that file [CMRH09]. In a similar line of work, Wolf et al. showed that build failures can be successfully predicted based on information representing the coordination of developers on shared work assignments [WSDN09]. In other research, a comparison of models for predicting fault proneness in Windows Vista binaries demonstrated that models composed of metrics based on developers and their arrangement in the organizational structure outperformed models composed of metrics based on source code (e.g, churn, complexity) and structural dependencies [NMB08]. The influence of coordination requirements are not limited to product quality, but can also negatively affect developer productivity. In a study of the alignment between communication channels and coordination requirements, the authors found that if a coordination requirement between developers is paired with a corresponding communication channel between developers, productivity increases [CH13]. These studies and others indicate that considering the developers in light of the technical work they do can be far more informative than perspectives focused solely on the technical dimension.

Coordination and teaming As developers work together to complete their work assignments, a number of factors impact the likelihood of a successful and timely outcome. Through team work, members of the team accumulate experience about the task domain and about each other, which helps to develop shared mental models and a common knowledge base [FSE07]. The presence of familiarity between team members has shown to improve team performance, particularly on tasks that are high in coordination complexity [ESKH07]. Intuitively, one could reason that coordination requirements that span a team boundary—one which is physical or virtual—is more challenging than coordination requirements occurring within a single team, because of the difference between inter- and intra-team familiarity. Not only does this expectation make sense intuitively, it has been confirmed several times empirically [CKI88; ESKH07; HM03a; KS95]. Specifically, one study found that work items spanning team boundaries took two-and-a-half times longer to complete than similar work items involving only one team [HM03a]. On this basis, software projects stand to benefit from the identification of developer groups that persists over a period of time and that are familiar with one another so that tasks with high coordination complexity can be strategically allocated to those groups [CH13]. More generally, the positioning of coordination requirements relative to an organization contains valuable information to support this kind of strategic task allocation approach.

Evolving coordination Throughout their lifetime, software systems are expected to undergo changes to their artifacts and artifacts' dependency structure. In terms of Lehman's first law of software evolution [LR03], "E-type systems must be continually adapted else they become progressively less satisfactory." As a software system evolves, particularly with respect to the system component dependencies, the need for developers to coordinate can be affected. Even minor changes to the architecture can create significant coordination challenges for the organization because of the impact the architectural change has on task interdependencies [HC90]. While it is still a major topic of discussion, some researchers believe that it is beneficial for software projects to align organizational structure with product structure [CH13; SER04], which is an idea originating from Conway's law. Conway's law, which in fact is more of a conjecture than a law, is the idea that organizations are constrained to produce designs that are copies of the communication channels between elements of the organization [Con68]. The implication is that a change in the product structure should be met with a corresponding change in the organization to avoid socio-technical misalignment between the organization and the architecture [BMB+07; SER04]. Consequently, to understand the nature of coordination requirements completely, a static view point is simply insufficient because it would neglect the intrinsic dynamic nature.

In summary, principles of modular design are critical to achieving a successful implementation of a complex software system in a large-scale globally-distributed project. A modular design will help support concurrent work assignment, but modularity as an approach on its own is insufficient. Regardless of how perfectly a system is decomposed into modules, concurrent task assignment gives rise to coordination requirements between developers. Without sufficient awareness of coordination requirements and a correspondingly appropriate organizational structure to adequately manage coordination requirements, the project will likely suffer in terms of developer productivity and software quality.

2.2 Software-Project Analysis

Software development is inherently complex. In this ambitious endeavor, tens, or in some cases hundreds, of individuals work together on intellectually and technically challenging tasks to construct systems comprised of millions of lines of code. A successful outcome depends on a high degree of regularity, discipline, and order. In an attempt to tame the potential chaos of software development, developers use a number of systems to help coordinate their efforts, collectively referred to as *software repositories.* As a side effect of using software repositories, a vast quantity of data are generated that provide an opportunity to study several important facets of software engineering. For example, the source code itself is a collection of information-rich artifacts. The version-control system (VCS) stores detailed information regarding every change made to the software. Issue or bug trackers document the enhancements that need to be implemented and defects that need to be fixed. In open-source software projects, communication between developers occurs primarily on a publicly visible forum, creating yet another important source of data. By integrating the information across multiple repositories into a single coherent representation, important questions regarding socio-technical aspects of software development can be addressed.

Recently, researchers have begun to leverage the wealth of information contained in software repositories to analyze software projects. Unfortunately, the intention behind software repositories was originally dedicated to static record keeping, so there are often substantial technical barriers to entering this line of research. Fortunately, the effort invested in analyzing software repositories is often returned with valuable context because they contain information not only on the present state of a project, but also indispensable historical data. The present state of a project is often sufficient to identify problems and support prescriptive solutions for how to fix the problem. What is missing from this viewpoint is historical context to understand the sequence of events that are responsible for generating the project conditions that lead to the problem. The historical context may be critical for uncovering the root cause of many problems that are faced during software development. We now introduce the data sources in more detail and discuss aspects of these sources that will play key roles in the development of ideas and techniques in the chapters that follow.

2.2.1 Source Code

There are numerous artifacts that are useful for analyzing a software project (e.g., requirements documents, design specifications, class diagrams, use cases). Source code is special in this regard because it is the only artifact that is guaranteed to be produced (barring project cancellations) and so it often takes a central role in software-project analysis. Source code also lends itself well to analysis because it is written with the intention to be complied or interpreted automatically, and so it adheres closely to a strict syntactic format that is easier to analyze through automated means than other types of artifacts designed with the primary intention to be human readable. As is the case for many complex systems, the elements of a software system can be grouped together according to various granularities. Researchers have dedicated significant effort to identifying sensible ways to group together the constituents of a software systems, and this plays a critical role in the ensuing analysis and resulting insights. Source code elements are typically grouped together according to the following categories.

- **System:** At the highest level of granularity there is the system level. This level encompasses the aggregate of every source code artifact that comprised the entire software system. Analysis techniques that operate at this granularity are often simple to design and implement because there is no need for sophisticated algorithms to identify which source code correspond to the system level. Insights generated by project analyses performed at the system level are helpful for high-level overviews and crude comparisons between different projects (e.g., counts of developers, files, changes, lines of code) but have limited insight into more detailed aspects of the project [LRW^{+}97; MFH02].

- **Subsystem:** Moving to the subsystem level takes a step towards a more fine-grained view on the project. The subsystems are typically a collection of files that are related at a high-level abstraction of the system. For example, some of the subsystems of the Linux Kernel are: Networking, Filesystem, and Memory management [Mau08]. The subsystems are typically inspired by the conceptual architecture and are reflected (although at times only imprecisely) by the project's directory structure. The range of sophistication for localizing source code to their corresponding subsystems varies from simple, by considering a subsystem to be a folder in the top level directory of the project, to complex, using sophisticated reengineering techniques to extract higher-order abstractions from the dependency structure of lower-level artifacts [MTW93].

- **Source Code File:** A source-code file represents yet another step towards finer granularity. In many programming languages (e.g., Python, C/C++, or Java) the abstract notion of modules is often mapped to individual files that contain the definitions for functions, methods, classes, and variables. According to principles of modular programming, a good system design involves modules with high cohesion and low coupling [AKC01; Par72]. On this basis, it is reasonable to expect that the lines of code within a single file are more interdependent than the code located in separate files. Given their nature, files are easy to identify and access, so localizing source code to its corresponding file does not require sophisticated approaches.

- **Source Code Entity:** In our final step towards finer granularity there is the level of source code entities. The set of source-code entities that can be defined depends on the given programming language. Most programming languages include the realization of a subroutine (e.g., a function) to group related code together that serves a single relatively narrow purpose. In object oriented languages (e.g., C++, Java, Python), entities include classes and methods. Identifying source-code entities is more complex than identifying source-code files, and relies to some extent on parsing and analyzing the source code. A number of static code analyzers exist for this purpose. If multi-language support is crucial and only basic information regarding the starting point of source code entities is required, the open-source tool Exuberant Ctags[1] provides sufficient functionality [Hie]. Doxygen is another open-source tool supporting many of the popular programming languages and provides both the start and end points of the source code entities [vHee].

Software projects can be analyzed at any of the above granularities. The appropriateness of the chosen granularity for a particular analysis is based on the characteristics of the phenomenon to be analyzed and the goals of the analysis results. By choosing the incorrect granularity, details of practical importance about the project can be concealed. For example, consider that the phenomenon being analyzed is source-code quality operationalized by the presence or absence of known bugs in the source code. At the system-level granularity, we may determine that there are N bugs in the system. On this basis, we decide to increase testing resources on all source code to try and bring the presence of bugs down. Alternatively, if bugs have been localized to the source code at the file-level granularity, the important insight that defects are non-uniformly distributed across the system would have been clear [FO00]. It is common that faults are described by a Pareto distribution.[2] With this information, we could strategically focus the testing resources on the small number of files that are likely to contain the majority of the bugs. In essence, the choice to use a finer granularity revealed important details regarding the statistical distribution of bugs, which was concealed by the system level analysis results.

[1]`http://ctags.sourceforge.net/`

[2]In this case, the data described by a Pareto distribution implies that most of the faults are contained in very small number of files.

2.2.2 Version-Control Systems

The version-control system is the primary tool for tracking modifications made to the software. Although, there is a number of different version-control systems, but in general, these systems all record the additions and deletions made to the software along with a collection of meta data about the modification. The VCS makes it possible to reconstruct the state of the software at any point along its development history. Git is a widely deployed distributed version-control system and we limit the following discussion of VCS specifics to Git [Tor]. In general, most VCS variants function in a similar manner with the primary differences arising from whether it is distributed or centralized [dAS09].

The unit of change in a VCS is a *commit* and is comprised of the following elements.

- **Commit Hash:** Each commit is assigned a unique identifier, which also corresponds to global revision numbers. The commit hash can be used to checkout the version that corresponds to the state of the software immediately after applying the changes in a given commit.

- **Author:** The author information typically includes the first and last name and the e-mail address of the individual responsible for authoring the changes.

- **Author Date:** The author date is a timestamp for when the author wrote the code contained in the commit and includes the date, time of day, and the author's time zone.

- **Committer:** In open-source projects, it is common that one individual writes code that is submitted in the form of a patch that is then applied (i.e., committed) to the software by a second individual on the author's behalf. The second individual is called the committer and is typically responsible for checking the quality of the change to make sure that it satisfies the guidelines specified by the project. The committer information includes the committer's first and last name and e-mail address.

- **Commit Date:** The commit date format is identical to the author date format but specifics the time when the patch is applied (i.e., committed) to the software by the committer.

- **Commit Message:** In the commit message, the author has an opportunity to describe the changes that are made by the commit in natural language. The commit message is typically very concise and gives a global overview of what the change included, the type of change (e.g., bug fix, enhancement, new feature), and why it was necessary.

- **Sign-off Tags:** A commit message can include a reference to a number of individuals other than the author and committer (e.g., testers or reviewers) in the form of Sign-off tags. Sign-off tags allow these other individuals to document the nature of their participation on a change. If a project is disciplined with including the sign-off tags, they can provide valuable information regarding roles and responsibilities of individual in the project.

- **Files Changed:** The filename and full path of any file that is affected by the changes introduced by the commit are documented.

- **Lines Changed:** For each commit, a diff can be generated for every changed file that expresses all added and deleted lines between two revisions.

In addition to the information contained in the VCS for analyzing software evolution, there are valuable data linking the software to the individuals responsible for it. More specifically, for any point along the development history, it is possible to determine the state of the source code and a reference to the responsible individuals down to the line of code granularity. In Git, the feature that provides this information is called “blame”. This feature is particularly useful for studying socio-technical aspects of software developer because connecting people to the technical work they do is a critical piece of information.

When analyzing VCS data, it is often necessary to group commits together to study the collective changes as a set. There are two primary reasons for grouping commits together. One reason is that processing the projects on an individual commit basis is computationally intensive because of the need to reconstruct the state of the project before and after every commit. By grouping the commits together, only the project state for the first and last commit need to be reconstructed. The disadvantage is that some information can be lost. For example, if a line of code is changed multiple times, only the last change is identified. This concern can be mitigated by ensuring the commit range is not too large (i.e., a range that defines a set of commits to include). The second reason for grouping commits together is to capture natural cycles that are present in the project’s development. For example, the Linux Kernel employs a number of phases, initially a *merge window* is opened to focus on the addition of new features, followed by a phase of stabilizing accepted features, then only bug fixes are made, and finally a release is cut (i.e., marking the end of a cycle) once the software it is deemed sufficiently stable [Mau08]. For some project analysis, it may be of interest to group together commits that correspond to these phases or all the changes that constitutes one complete development cycle.

To group commits together, a window or range must be defined to specify how the commits should be grouped. At a high level there are two approaches to specifying the grouping of commits:

- **Revision-based:** In the revision-based approach, a range is specified by a pair of revisions. For example, if the range is specified by (Rev$_1$, Rev$_2$), the collection of commits are those that occur after Rev$_1$ and not after Rev$_2$. More technically, it is the commits that are reachable from Rev$_2$, but not reachable from Rev$_1$.[3] In some cases, there are revisions that correspond to important events or project milestones. Git offers a tagging feature to reference these revisions with special tag. These tags can then be used to analyze a project based on a collection of commits that correspond to project milestones. Unfortunately, projects often use tags differently or sometimes without any particular regularity or meaning. This fact can make it difficult to make sensible comparisons with projects that have been analyzed on a revision bases.

- **Time-based:** In the time-based approach, commits are grouped according to calendar time by specifying a pair of dates. For example, if the range is specified by (Date$_1$, Date$_2$), the collection of commits are those that contain an author date (or commit date, depending on the configuration) that occurs after Date$_1$ and not after Date$_2$. The benefit to using a time-based approach is that a date range has the same meaning regardless of the project it is being applied to. For this reason, it is sensible to use a the time-based approach for performing project comparisons. The challenge with applying a time-based approach is to determine an appropriate time-window size.

In Figure 2.2, a linearized history of nine commits is shown with edges between commits expressing parent-child relationships. Three revision tags mark the point at which revisions were cut at commits C_1, C_5, and C_9 corresponding to revision Rev$_1$, Rev$_2$ and Rev$_3$, respectively. Calendar time references are represented as dotted vertical lines separated by one month for a three month period of the development history. In a revision-based approach for the range (Rev$_1$, Rev$_2$), the commits selected for analysis are $\{C_2, C_3, C_4, C_5\}$. In a time-based approach for the range January – February, the commits selected for

[3]Commits in the VCS are arrange as a directed acyclic graph where nodes are commits and edges point to the parent of the commit. Reachable in this sense means the commit exists on some path (by traversing the graph without violating edge directionality) with a specified origin.

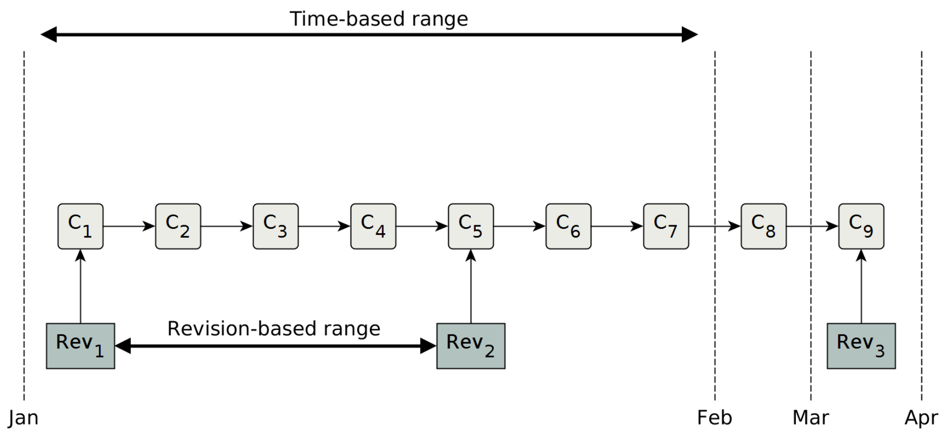
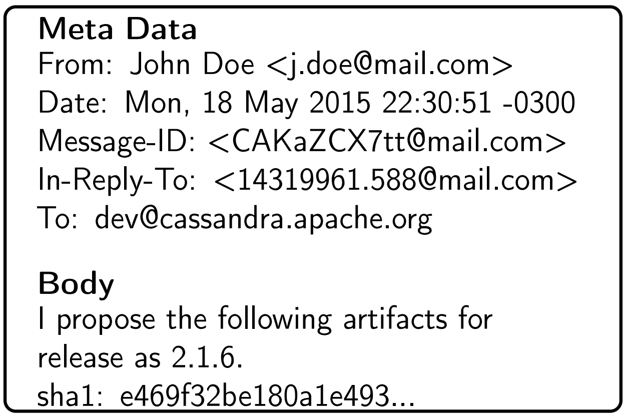


Figure 2.2: A linearized history, three months in length, is shown containing nine commits. Two revision tags indicate which commits mark the end points for three major revisions, Rev$_1$, Rev$_2$, and Rev$_3$. The notion of a time-based range and a revision-based range are represented.

analysis are $\{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$. Based on time, it is clear that January–February was a uniquely productive or active period compared to the following months. From the revision-based range, this insight is missed because the revision cycles are not necessarily related to calendar time but rather to project specific details (e.g., completing milestones). The benefit of using a time-based range is that all projects, at all times, are subject to the same units of calendar time, but what constitutes a new revision is more ambiguous. When considering the activities of developers, we also need to remain cognizant of the impacts of temporal proximity in the activities. If the activities are separated significantly in time, the relationship or interpretation of those events may be different. For example, if two developers work intensively on a single artifact over a short period of time, this may imply a strong need for the developers to coordinate their work to ensure that the concurrent development does not lead to changes that interfere with each other in unexpected ways. If the changes made by each developer are instead separated by two years, the nature of the event is different since it is less likely for their changes to have unintended side effects. The later developer is able to see all of the initial developers changes, and introduce new changes to the artifact without as much uncertainty. Essentially, the nature of concurrent development and sequential development has different implications on the need for coordination and to differentiate between these two cases temporal considerations are of utmost importance.


Meta Data
From: John Doe <j.doe@mail.com>
Date: Mon, 18 May 2015 22:30:51 -0300
Message-ID: <CAKaZCX7tt@mail.com>
In-Reply-To: <14319961.588@mail.com>
To: dev@cassandra.apache.org

Body
I propose the following artifacts for
release as 2.1.6.
sha1: e469f32be180a1e493...

Figure 2.3: Example of the information that is contained in a single message that is posted to a mailing list.

2.2.3 Mailing Lists

In closed-source or commercial software projects, most of the communication is accomplished via private e-mail or messaging services, face-to-face discussions, or phone calls. Instead of these communication mediums, open-source projects typically use public forums called *mailing lists* as the primary means of communication between individuals [BPD+08; DOS99]. On a public mailing list, anyone can view the messages that are exchanged and anyone can post a message for everyone else to view and respond to. The participants of open-source project mailing lists include developers, bug reporters, and users. Often projects host multiple mailing lists to separate technical development topics from user related questions.

In mailing lists focused on technical discussions many of the contributors are also developers or have had substantial prior experience as a developer on the project. The discussion topics—while primarily technical in nature—serve a variety of purposes. Some projects have very strict policies that make it mandatory for design decisions, proposals for features, and any source-code changes to first be posted to the mailing list, where they can be discussed openly before making final decisions [Cor]. For example, if someone has found a bug and fixed it, they can submit a patch to the mailing list. At this point, members of the community have an opportunity to comment on the patch and either approve it or make recommendations for improvement. In this regard, the mailing list is a fundamental element of the governance process supporting the broader philosophy of open-source development, which dictates that all suggestions for change should be open to public scrutiny and only then will the best ideas prevail [DOS99]. The mailing lists of open-source projects contains a wealth of information regarding the communication channels that exists between developers. The contents of a mailing list can be obtained as a set of mails in a plain text file format referred to as *MBOX*. The mail format contains the message content in addition to several pieces of meta data. Instead of giving a complete overview of the mail format, we focus on the details that are most relevant.[4] An example message is shown in Figure 2.3, and a description of each field is given below.

- **From:** The "From:" field contains the name and e-mail address of the individual responsible for writing the message. In some cases automated services send e-mail notifications to the mailing list (e.g., Github will send mails for activity in the VCS), so people are not responsible for all messages. This needs to be considered when analyzing the mailing list to avoid overestimating developer activity.

- **Date:** The "Date:" field contains the timestamp for when the mail was sent by the author. This information includes the date, time, and local timezone.

- **Message-ID:** The "Message-ID:" field contains a unique identifier for each message. This field is designed to be machine readable so that a chain of messages can be automatically reconstructed by using this field to cross reference.

- **In-Reply-To:** The "In-Reply-To:" field contains the Message-ID for one or more messages. When a message M_1 is sent in response to another message M_2, the "In-Reply-To:" field of M_1 contains the Message-ID of M_2. In the case of multiple reply messages, the Message-ID of all prior messages is stored as a list in the "In-Reply-To:" field. For example, in the case of the message depicted in Figure 2.3, any future message containing the value "In-Reply-To: <CAKaZCX7tt@mail.com>", is recognized as pertaining to a common thread of communication.

- **To:** The "To:" field contains the e-mail addresses of the intended recipients of the message. In the case of mailing lists, this is the e-mail address of the specific mailing list (e.g., the mailing list for technical discussion or a user oriented mailing list).

- **Body:** The message body contains the primary textual content written by the author of the message.

[4]For a complete technical documentation of the Internet Message Format specification, please refer to: `http://www.rfc-base.org/rfc-5322.html`.

Individual messages contain useful information in that they connect people to the content they write about, but that does not reveal a lot about the community or groups of individuals. To capture information about the community, we first need to understand the mechanism behind initiating messages. On a mailing list, if an individual has a topic they want to discuss, and this particularly topic is not currently already under discussion, the individual will post a new message directly to the mailing list by sending an e-mail to the mailing list address. Everyone who has already subscribed to the mailing list will receive a notification of this new message. If this new topic is interesting for someone, they may wish to post a message in reply to the original message. By tracing the Message-IDs that appear in the "In-Reply-To:" field of the messages, a thread of communication can be reconstructed from the MBOX file. The assumption is that the series of messages that appear in the "In-Reply-To:" field have a relatively narrow focus and the individuals contributing to a common thread are not only aware of each other, but also share a common interest. This assumption is reasonable because many open source projects rely on the mailing list as a form of documentation for frequently asked questions. For this reason, projects request that threads remain focused on a single topic to make it easier for others to identify relevant information based on the thread's subject line and benefit issues that have already been discussed. By grouping mails together based on the thread concept, one is able to gain further insight not only on individuals, but also at a community level. For example, the number of messages composing the thread and number of unique individuals contributing to the thread may indicate the relative importance of a topic to the community. Topics with many responses from a varied set of individuals is likely to be highly relevant or interesting to the project and its contributors.

2.2.4 Mining Repositories

The discipline of mining software repositories has emerged in response to the challenges and rewards of analyzing data in VCS, issue trackers, and mailing lists. A wide variety of techniques have been established to elicit meaningful and actionable insights from software repositories to improve software development by using data-driven reasoning [KCM07]. Developing these techniques is challenging because the data in software repositories is often unstructured and can suffer from low data quality (e.g., data consistency and integrity). The VCS can have missing meta data, a rewritten history that does not reflect the original commit sequence, or a non-linear commit ancestry making a linearized view difficult or impossible without losing information [BRB$^+$09]. In mailing lists, messages can be duplicated but contain different Message-IDs and incorrectly formatted fields that deviate in unexpected ways from the RFC 5322 format specification.

In many applications it is necessary to attribute all activities performed by one individual, across multiple repositories, to a single unique identity. To give a concrete example, if a developer makes commits to the VCS and enters into discussions on the mailing list, we would like to resolve these two activities to the same individual. This way, we are able to address questions that span the socio-technical boundary like, do developers making lots of contributions to the code base also communicate a lot on the mailing list? There are however technical problems with this because, in general, there exists no formal reference between the contributions made by single individual to multiple repositories. For this reason, string matches on names and e-mail addresses are often the basis of an imperfect solution to this problem. The solution is inadequate in some cases because a developer may use multiple aliases or change their identify over time. For example, LLVM developer Greg Bedwell has used the E-mail address greg_bedwell@sn.scee.net and gregbedwell@gmail.com to make contributions to the project. For this reason, error tolerant string matching using a probabilistic approach is needed to resolve the identify to a single individual. More sophisticated error tolerant string matching approaches that make use of the Levenshtein edit distance have shown improvement over the basic string match approach, but still fail under some circumstances [BGD+06].

To make practical steps forward in this domain of imperfect data sources, a set of assumptions is often necessary. For example, in the analysis of mailing lists there is often the assumption that a thread of communication has a relatively narrow and stable focus [BGD+06; BPD+08; PBD08]. Though, in reality, a thread may exhibit concept drift and diverge from its original purpose, especially when a thread involves many messages over a long period of time. The same is true for approaches applied to the VCS. For example, there is often the assumption that a file contains code that is highly interrelated, though, in reality a file may provide a wide variety of utilities. In abstract terms, there are two important considerations to be made. Firstly, because of imperfect data, practical assumptions will often fail to cover 100% of cases, that is, completeness is rarely achievable. As with the case of aliases, by taking on a probabilistic approach, we can build in error tolerance to help avoid making overly strict assumptions. Probabilistic analysis approaches are a way to mitigate the negative consequences of our assumptions by providing the foundation to systematically reason about data sources that are noisy and imperfect. From this perspective, probability theory is an extended theory of logic that incorporates the ability to reason about situations with inherent uncertainty or where only incomplete knowledge is available. Secondly, any assumptions being made in the approach need to be tested for validity and met with a sufficient degree of skepticism and attention to detail. Testing for validity is often challenging in this discipline because there is often no widely accepted ground-truth, and if one exists, it is often not easily attainable. If the goals of the analysis are to model real-world phenomena, then one option is to test how well insights derived from a model are consistent with the real-world. In the case of mailing list analyses, one goal is to establish accurate knowledge of which developers are aware of each other and the tasks they work on. A reasonable test of validity would be to ask developers if the individuals we expect them to be aware of, based on software repository mining techniques, agree with their perceptions. The outcome of such a test would be how accurate is the approach, under which conditions does it succeed or fail, and lead to new refinements on the approach that increase the validity of insights. For example, we may learn that extremely long mail threads have likely diverged and so we eliminate threads with more than X number of messages when inferring relationships between developers.

2.3 Network Analysis

The term “network” is used to denote a relational perspective on data. However, the term is used in a large variety of domains (e.g., electrical circuits, communication networks, transportation systems, bioinformatics, etc.), so we explicitly state our focus on networks to their application for modeling complex systems. From this it follows that “network analysis” is the collection of techniques used to study the structure of complex systems, which are formalized in a network representation. The ubiquity of networks is often referenced as a testament to their importance and relevance with phrases like "networks are everywhere." It is our intention to emphasize that networks are often an abstraction of a real-world phenomenon, and that networks do not simply appear out of nothing. Rather, the abstraction is accompanied by a set of nontrivial steps that should be made with a degree of delicacy and thoughtfulness. The construction of networks is not the primary focus of network analysis, but it is important to maintain a critical stance toward the steps that were taking prior to performing analysis on the network. If the steps taking to abstract a phenomenon into a network representation are not sensibly made, no level of sophistication in the network analysis will elicit meaningful insights.

2.3.1 A Network Perspective

Before delving into the details and formalizations of networks and network analysis, we take a short detour to investigate the choice of using a network perspective. While there are certainly benefits to representing a set of data as a network, it should be noted that complex networks are often very high dimensional structures, which presents significant challenges in terms of algorithmic complexity. It is often the case that only algorithms with exponential complexity are available. Even for extremely fundamental mathematical operations, such as testing two networks for equality (i.e., graph isomorphism), only nonpolynomial time algorithms are known today. For this reason, a justification that outweighs the mathematical convenience of alternative representations is necessary to rationalize the choice to model data as a network. To illustrate the benefits of a network perspective, we make a brief comparison between standard data representation and network data representation to emphasize the fundamental difference.

In the standard representation, the unit of observation is an entity with corresponding attributes. For example, the entity “vehicle” has attributes that express the vehicle’s engine size, efficiency, and color. For each entity in the set, all the same attributes are measured and each entity is assumed to be independent. Essentially, there is no structure expressed between the entities, the collection is simply treated as an unordered set of observations. There is even the often explicit condition that the observations are statistically independent. Independence is this case means that the sequence of observations is equally likely to appear as any permutation of the observed sequence, which justifies treating the observations as an unordered (i.e., unstructured) set. Any dependence between entities is typically viewed as a nuisance relationship that should be eliminated. For example, the observation of a red vehicle is not expected to impact the probability of observing any particular colored vehicle in the following observation. Typically, goals of the analysis are primarily concerned with examining relationships between attributes of the entities, not the entities themselves, like a vehicle’s efficiency and engine size. In this analysis, we want to avoid any relationship between the entities that may influence the relationship between attributes efficiency and engine size (e.g., many vehicles in the sample are hybrid electric).

In contrast, a network is defined on a domain of overlapping dyads. In the network representation, there are still entities and attributes, but there are now also ties to express relationships between entities. In a scientific co-authorship network, ties indicate that the entities have both participated in authoring a single scientific publication [POM10]. Here, the unit of observation is pairs of entities, where each entity can appear in multiple dyads. A single author may participate in authoring papers with a number of different individuals on a number of different papers. The network representation is inherently focused on within-variable associations. If the within-variable associations are totally independent, then there is not much benefit to a network perspective. In the case of scientific authorship, typically groups of scientists form around topics, resources and shared interests, so there are meaningful insights that are a product of correlations between the entities of a co-authorship network that give rise to structure [POM10]. In the evolution of affiliation networks, a common process is that of *triadic closure*, where there is a bias for edges to form between entities that have a mutual acquaintance[KW06]. This is once again draws attention to the emphasis on within-variable dependence that is observable from a network perspective. When we later look at random networks, we will see that several meaningful network organizational principles that arise from interdependence between ties. We will see that the dependence among ties is the mechanism that gives rise to self-organization and patterns of evolution in the network. Tie dependences also explains the structure of many empirical networks, so it is not just an academic curiosity.

In essence, when there is interdependence between entities, a network view can provide an opportunity to explore the rich structure that emerges from those interdependent relationships. Where in the standard representation, entity interdependence was a nuisance factor, in the network representation, entity interdependence is the primary interest. In the following subsections, several different notions of organized structure that emerges from non-random entity interdependencies are introduced and formalized.

2.3.2 Graphs

So far, we have spoking of networks in abstract terms as a collection of entities that can exhibit ties between the elements to form dyads. The notation of graphs fits very naturally to this conceptualization and so we make heavy use of graph notation to formalized the following network analysis concepts.

Formally, a graph $G = (V, E)$ is composed of a finite set V of vertices (nodes) and a set E of edges (links), denoted by $V(G)$ and $E(G)$, respectively. The edge set is generated by linking vertices according to $E \subseteq V \times V$. Edges are defined by pairs of vertices u and v, so that $u \in V$ and $v \in V$. In a *directed graph*, the edges are specified by ordered pairs $\langle u, v\rangle$ where u is considered to be the source vertex and v to be the target vertex. In an *undirected graph*, the edges are specified as sets so that $\{u, v\} = \{v, u\}$. If two vertices are connected by an edge, they are considered to be *adjacent* vertices or *neighbors*. The collection of neighbors of a vertex v is denoted by $N(v)$. In a *weighted graph*, edges are assigned a weight according to a function $w : E \to \mathbb{R}$, which assigns to each edge $e \in E$ a real-valued weight $w(e)$. For *unweighted graphs*, edges it can be considered a special variant of a weighted graph where $w(e) = 1$ for all $e \in E$. The interpretation of the edge weight is specific to the domain of application. In some cases, the weight can indicate the strength of relationship between the vertices or the distance between them.

A commonly used and mathematically convenient way to express a graph is in matrix form. The adjacency matrix $A = (a_{i,j})_{n \times n}$ for a graph $G = (V, E)$ is an $N \times N$ matrix where $N = |V(G)|$ defined by

$$a_{i,j} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

such that elements of the adjacency matrix are assigned according to $A_{i,j} = 1$ if vertex v_i and vertex v_j of graph G are adjacent and $A_{i,j} = 0$ if the vertices are not adjacent.

A graph can be divided into subgraphs that represent a portion of the original graph's topology. The graph $G_1 = (V_1, E_1)$ is a subgraph of the graph $G_2 = (V_2, E_2)$ if $V(G_1) \subseteq V(G_2)$ and $E(G_1) \subseteq E(G_2)$. In the more strict case where $E_1 = E_2 \cap V_1 \times V_1$, G_1 is referred to as the *vertex-induced subgraph* of G_2.

2.3.3 Vertex-Level Metrics

There are various metrics that quantify some property of a vertex relative to its local or global environment. Centrality metrics assign a value to each vertex based on its relative position in the graph. The simplest of the centrality metrics is *degree centrality* and is defined to be the number of edges in E that involve a given vertex v. For a vertex v, degree centrality is denoted by $\deg(v)$ and measures local centrality, because it is computed using only knowledge of the immediate neighbors of v. In Figure 2.4 the degree centrality for the filled and unfilled vertices is four and one, respectively. In the case of a directed network, degree centrality is split into two values, where *in-degree* is the number of edges terminating at v, and *out-degree* is the number of edges originating from v. In the case of a weighted graph, degree centrality is computing by summing over the relevant edge weights. In another variety of centrality metrics, the centrality measure indicates not simply how central a vertex is based on the local neighborhood, but rather on the centrality of the vertices in the local neighborhood. In other words, the centrality of vertex v is defined recursively based on the centrality of the neighbors of v. To illustrate the intuition, consider that a president of a country is not typically important by virtue of direct contact to an enormous number of subordinates (i.e., a high

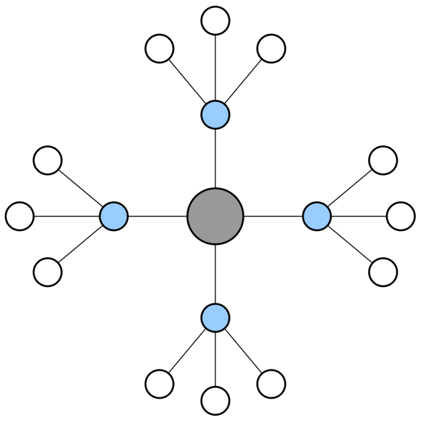


Figure 2.4: A network comprised of 17 vertices and 16 edges. The degree centrality for all filled vertices is four. The larger central vertex intuitively appears more important than the other filled nodes, despite the equivalent degree centrality. Eigenvalue centrality assigns the highest centrality to the larger node (center) because all it's neighbors exhibit relatively high degree centralities (each neighbor has degree four). In comparison, The smaller filled nodes are each connected to three nodes with degree one and a single node with degree four.

degree centrality). Instead, the importance of a president stems from the fact that the few subordinates that are directly in contact with the president are highly important individuals. A minimal example of this intuition is provided in Figure 2.4, where the president is depicted as a large filled vertex at the center connected to subordinates represented by smaller filled vertices. Eigenvector centrality is a commonly used variant of centrality that captures this notion of vertex importance in a network. The basics of eigenvector centrality are most easily explained in terms of the adjacency matrix representation of a graph. The eigenvalue centrality for vertex i is then represented by x_i according to

$$x_i = \frac{1}{\lambda} \sum_{j \in N(i)} x_j, \tag{2.2}$$

where $N(i)$ are all direct neighbors of vertex i and λ is a proportionality constant. This can be rewritten using the adjacency matrix as

$$x_i = \frac{1}{\lambda} \sum_{j=1}^{N} A_{i,j} x_j, \tag{2.3}$$

and in matrix notation the summation can be rewritten as $\mathbf{x} = \frac{1}{\lambda}\mathbf{A}\mathbf{x}$. From this point the equation can be recognized as eigenvector equation $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$.

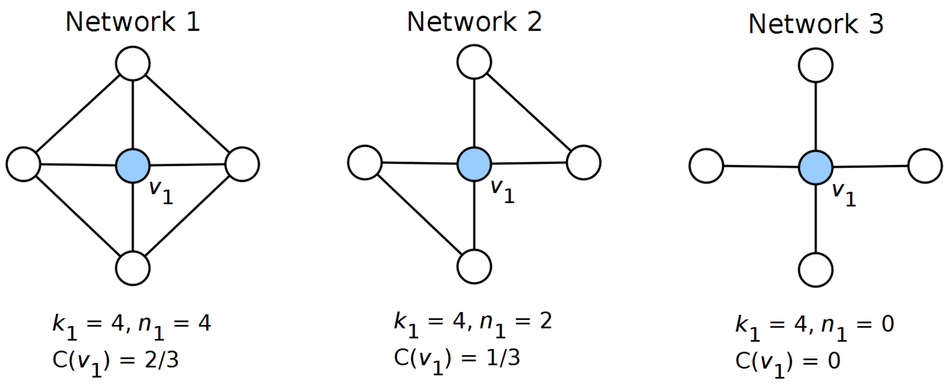


Figure 2.5: Three networks illustrating decreasing clustering coefficients (left to right) for the filled node. In network 1 the clustering coefficient is highest, because many neighbors of the filled node are connected. The neighbors of the filled node in network 2 are less connected compared to network 1, thus reducing the filled node's clustering coefficient. In network 3, none of the neighbors are connected and therefore the clustering coefficient is zero.

A unique solution to this equation is guaranteed by solving for the largest real eigenvalue corresponding to a strictly positive eigenvector [Bon87]. The ith component of the eigenvector then corresponds to the centrality score for vertex i. Google's PageRank algorithm for ranking the importance of a Web page based on its centrality within a network composed of Web pages (the vertices) and hyperlinks (the edges) is fundamentally based on the principles of eigenvector centrality [Fra11]. The key differences are that PageRank computes a probability that a random walk visits a given vertex, and a damping factor is used that corresponds to the probability that the random walk terminates and begins again on a randomly selected vertex [PBMW99].

clustering coefficient

Beyond centrality metrics, a vertex can be assigned a quantity based on the local neighborhood connectivity. The *clustering coefficient* is an example of a metric of this type. The clustering coefficient represents the likelihood that neighbors of a vertex are also connected to one another. Essentially, the metric indicates to what extent a vertex is embedded in a densely interconnected set of vertices, called a cluster. In this respect, the clustering coefficient is an indicator of the modularity. The vertex clustering coefficient is defined as

$$c_i = \frac{2n_i}{k_i(k_i - 1)}, \tag{2.4}$$

where c_i is the clustering coefficient of vertex i, k_i is the number of vertices adjacent to vertex i (i.e., $k_i = |N(i)|$), and n_i is the number of edges between the vertices adjacent to vertex i. The intuition is that $k_i(k_i - 1)/2$ number of edges can exist between k_i nodes, and the clustering coefficient is a ratio that reflects the fraction of existing edges between neighbors divided by the total number of possible edges. In Figure 2.5, three example networks are shown in which the filled vertex has a decreasing clustering coefficient from left to right.

2.3.4 Random Network Theory

The goals of random graph theory strive to explain the properties of a graph, or a graphs generative process, in terms of probabilistic notions. Interestingly, many important topological properties of real-world networks are described by incredibly simple probabilistic models, which makes probability theory an attractive tool to reason about the complexity of large real-world networks. Initially, Erdős and Rényi proposed a very simple model to describe the generative process of a complex network, which produces the so-called ER random graph. In this model, a set of n vertices is specified and the probability of an edge occurring between any two vertices is fixed with a probability p [ER59]. This initial model captured some interesting properties, such as that the shortest path between two randomly selected vertices grows proportionally to the logarithm of the number of vertices in the network, a property referred to as the small-world property. The ER random graph served an important purpose in triggering a substantial interest in the field of random graph theory, but was later shown to have limited use because it failed to explain many other fundamental properties of real-world networks.

degree distribution

The degree distribution of a graph is a statistical property that has important implications on the graph's topology. Recall that the degree of a vertex, denoted by $\deg(v)$, is the number of edges incident on vertex v. The degree distribution for given graph is defined as, $\Pr(\deg(v) = k)$, which represents the probability of observing a vertex v with a given degree k. In the case of ER random graphs with a large number of nodes, the degree distribution converges to a Poisson distribution (a member of the exponential family). Contrary to this result for ER random graphs, many real-world network exhibit degree distributions described by a class of heavy-tailed distributions [DM03].[5] This observation triggered significant interest to look for other plausible generative models that could explain the formation of a heavy-tailed degree distribution. In Figure 2.6, a comparison between exponential scaling, which occurs in ER random graphs, and power-law scaling, which occurs in many real-world networks, is shown. The heavy-tail of the power-law distribution leads to a fundamentally different

[5]A heavy-tailed distribution exhibits slower than exponential decay. This leads to significant weight in the tail so that the probability of observing extreme values is not negligible.

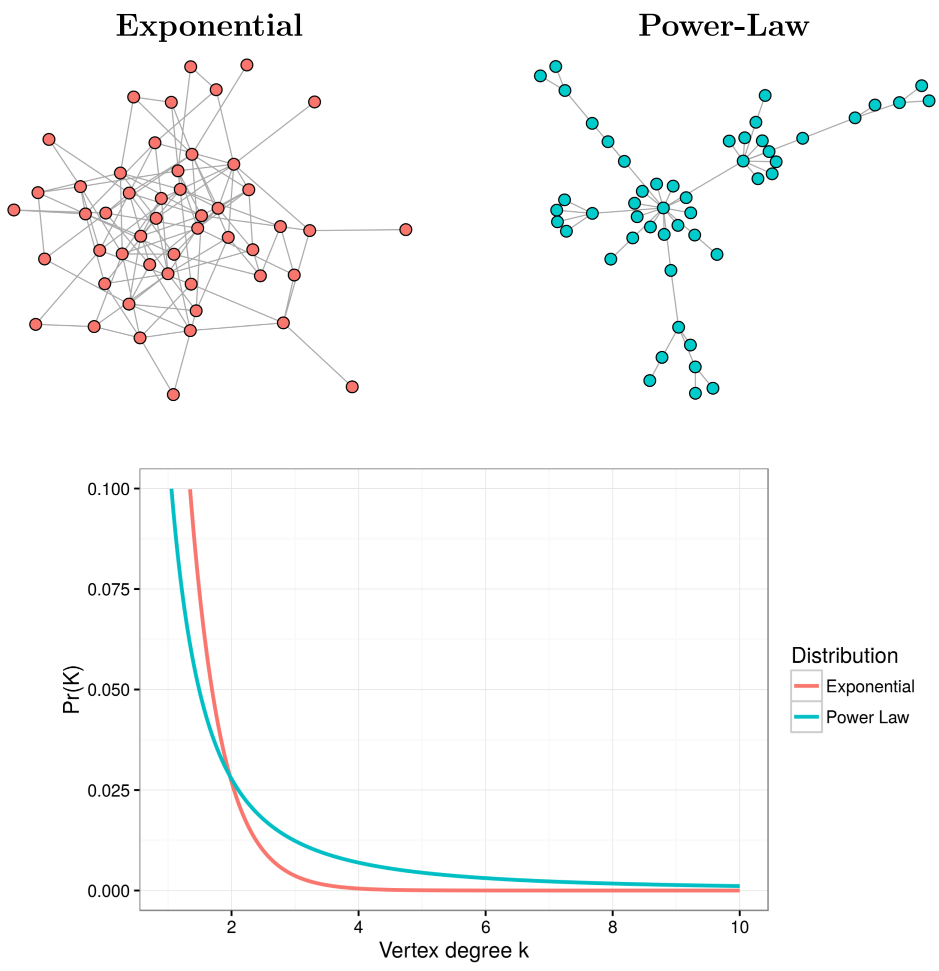


Figure 2.6: A network with an exponential degree distribution (top left), and power-law degree distribution (top right). The corresponding degree distribution for exponential $\Pr(k; \gamma) = e^{-\gamma k}$ and power-law $\Pr(k; \gamma) = k^{-\gamma}$ are show (bottom). The power-law distribution contains significantly more weight in the right tail compared to the exponential distribution. The heavy tail gives rise to the organized structure of the network with a power-law degree distribution (top right) where a small number of nodes are hubs and low degree nodes collect around these hubs.

network topology because it is far more probable to observe a vertex with a degree that is significantly larger than the mean. The right tail of the exponential distribution contains much less weight than the power law, so it is extremely unlikely to observe a vertex with degree much larger than 2.

preferential attachment

The implication of a heavy-tailed degree distribution is that a few vertices will form hubs that have a degree that is significantly larger than most other vertices in the network. Another way to view this is that a severe inequality, or bias, exists in the distribution of edges among nodes. In the ER random graph model, the probability that any node has an incident edge is fixed by the probability p, so that any inequality in the distribution of edges is a result of stochastic uncertainty rather than a systematic bias in the generative process. That is why in Figure 2.6, the network with the exponential degree distribution has a typical vertex, and the degree of other vertices deviate from the typical vertex only slightly. This is not the case in the network with a heavy-tailed degree distribution, where a small number of vertices are connected to the majority of edges. To explain this curious inequality in the distribution of edges among vertices, the model of *preferential attachment* was proposed [BA99]. In preferential attachment, there is a bias in the generative model that favors attachment of new edges toward vertices that already have a high degree. A variety of preferential attachment models exist that differ in how strong the bias is and by the functional relationship the bias follows (e.g., linear or exponential) [NBW06]. Here, we illustrate only a single model capturing linear scaling in the degree proposed by Barabási et al. [BA99], defined as

$$p_i = \sum_{j \in V} \frac{\deg(i)}{\deg(j)}, \tag{2.5}$$

where p_i is the probability that a new node is attached to vertex i, which is proportional to the degree of vertex i. According to this formulation, vertices with a high degree are more likely to receive the attachment to the new vertex then vertices with a low degree. The success of preferential attachment is largely a product of its ability to generate graphs that obey the same heavy-tailed degree distribution of many real-world networks [DM03].

configuration model

As we have seen, the degree distribution is a fundamental characteristic of complex networks by explaining the existence of certain topological properties and hinting at plausible evolutionary processes. For this reason, there is significant interest in generating graphs with a specified degree distribution, but uniformly random in all other aspects of how the vertices are connected. This approach is very useful to identify what topological properties are explained as a consequence of specific degree distributions [MKI+03]. For example, does a heavy-tailed degree distribution always lead to a low clustering coefficient and the formation of clusters of vertices that associate strongly with each other or is the low clustering coefficient a consequence of a separate organizational principle? It may be the case that an empirical network exhibits structures that stem from organizational principles that extend beyond what is a product of its degree distribution. This model allows us to generate a family of graphs, all of

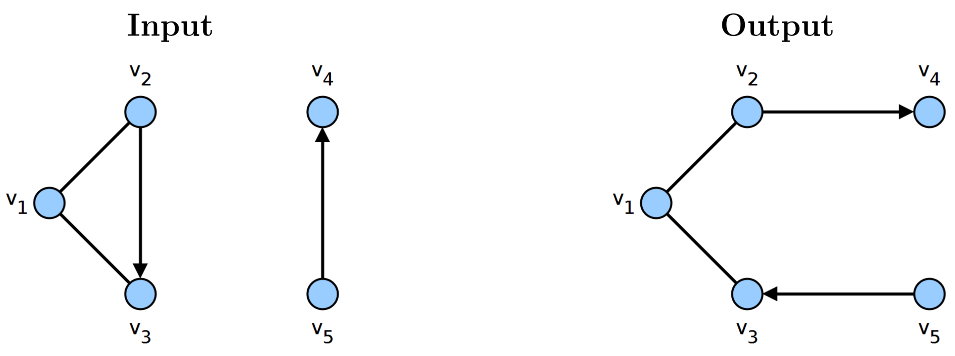


Figure 2.7: An input graph (left) containing five vertices with specified degree. After applying the configuration model to the input graph, an output graph (right) is generated containing the identical number of vertices and degree sequence. As a consequence of applying the configuration model to the input graph, the preference for $\{v_1, v_2, v_3\}$ to associate is destroyed by the randomization process.

which can exhibit different characteristics, but have the unifying characteristic that the degree distribution is held constant. The ER random graph model and the model of preferential attachment were parameterized based on a number of nodes and a function or constant to define the probabilistic mechanism for attaching nodes to each other. In this random graph model, the input parameter is a degree distribution or a degree sequence and is referred to as the *configuration model*. The configuration model plays in important role in generating null models from empirical networks that can be used to test for the presence of structural properties that are not explained solely by a specific degree distribution plus uniform randomness [MKI+03]. In Figure 2.7, an empirical network is shown as the input graph and the output graph is shown after applying the configuration model. Notice that the degree of each node remains constant between the input and output graphs, but now the two isolated subgraphs in the input graph are connected into one component. Algorithms for generating random graphs based on the configuration model are rather complex, however, intuitively, the process can be described rather simply. First, a sequence of vertices with their corresponding degrees are provided. Each vertex is then represented with a number of half edges (edges with only one vertex connected) corresponding to its intended degree. Uniformly at random, two half edges are chosen and joined together to form an complete edge. This process is carried out until there are no more remaining half edges.

2.3.5 Scale-Free Networks

The degree distribution of many real-world networks tend to have a heavy tail, which results in the likely formation of hub nodes (i.e., nodes with a significantly larger degree than the mean degree). It turns out that one type of heavy-tailed distribution appears to be nearly ubiquitous in nature: the Pareto distribution or power-law distribution. A network that exhibits a degree distribution that is described by a power law, formalized as

$$P(\deg(v) = k) = ak^{-\gamma} \qquad \gamma > 0, \tag{2.6}$$

is referred to as a *scale-free network*. The name "scale free" is a reference to the lack of characteristic scale exhibited by power-law functions. This property can be shown by scaling the input of by a multiplicative factor. Provided that $f(x) = x^{-\gamma}$ then

$$f(ax) = (ax)^{-\gamma} = a^{-\gamma}x^{-\gamma} = cx^{-\gamma} \propto f(x), \tag{2.7}$$

which demonstrates that, for this function, scaling the input x does not change the shape of the function because it exhibits scale invariance.

network robustness

Practical Implications Beyond the apparent ubiquity of power-law degree distributions in real-world networks (e.g., Internet, scientific paper co-authorship, power grid, biological systems), scale free networks have a number of practical implications [DM03]. For example, the failure characteristics of complex systems are of primary interest for domains such as the power grid or the Internet, where a catastrophic failure in the operation of these networks has widespread and costly consequences. The topology of a network determines the severity of impact from disruptions or perturbations to the network. A network perturbation manifests as an edge or vertex deletion, and different types of perturbations can be realized by altering the mechanism used to select an edge or vertex for deletion. In the case of random perturbations, the selection is made uniformly at random by assigning an equal probability of selection to all elements (edges or vertices) of the network. In a network comprised of a company's employees and edges expressing the organizational structure, a source of perturbations to this type of network arise from employee turn-over. Depending on the network's structure, the effects turn-over may impact the disseminate of knowledge in the company and hinder progress. In targeted perturbations, elements of the network are selected for deletion based on their location or function in the network (e.g., a highly central vertex). For the employee network, a source of targeted perturbations are layoffs, which are often intentionally localized to a specific department or team such that not every employee is subject to the same probability of losing their job.

Using simulation techniques, it has been shown several times that scale-free networks are incredibly robust to random perturbations, but extremely vulnerable to targeted perturbations. The results demonstrate that the path length between vertices in the network composed of Internet routers increases only slightly even when a large number of routers fail at random [DM03]. It is important to recognize that the cost of robustness to random failures is paid with susceptibility to targeted attacks. The point is that the claim of robustness as a benefit to scale-free networks is limited to specific failure conditions. If the main threat to a network is a small number of targeted attacks that are intentionally designed to inflict maximal destruction, scale-free networks present an enormous risk because the deletion of a small number of hub nodes can fracture a connected graph into disjoint subgraphs [CEbH01]. The overall message is that the optimal structure of a network is intimately connected to the environmental conditions in which the network operates. By studying networks, structural deficiencies or other weakness can be identified. It is with this knowledge that strategies can then be developed to most effectively compensate for weaknesses and maximize benefits from advantages. In the case of the Internet, optimizations could be realized by maximizing proliferation of the Internet by using cheap router hardware (where random failure robustness is exploited) and substantial security efforts made to protect hub routers (where vulnerability to tarted attacks is mitigated). For companies, the implication of a scale-free organizational structure implies that turn-over is generally tolerable but retainment of key individuals is absolutely critical.

Scale-free hypothesis test

Identifying Scale-freeness Testing for the presence a scale-free network in empirical data is a matter of invoking appropriate statistical machinery, but there are certain non-trivialities involved. The uninformed or inexperienced individual may conclude that linear regression forms the basis of a sufficient statistical test after applying a logarithmic transformation to the data. The rationale is that, since a power-law distribution is defined by $f(x) = ax^{-\gamma}$ then $\log(f(x)) = \log(ax^{-\gamma}) = -\gamma\log(ax)$, which is a linear model in terms of the model parameter γ. The problem with this approach is that many functions, not only power functions, can appear roughly linear (especially within a small range of values) after applying a logarithmic transform, so this criterion is necessary but insufficient [CSN09]. Another problem is that only in rare cases the degree distribution of a scale free network obeys a power law for all values. More often, there is a minimum degree $x_{\min}$, where for values greater than $x_{\min}$, the data is distributed according to a power-law distribution, but for values less than $x_{\min}$, the data is distributed differently. This means that the identification of a power-law distribution in empirical data amounts to estimating two parameters, $x_{\min}$ and the scaling parameter γ.

By using the method of *maximum likelihood*, one is able to estimate the model parameters on the basis of maximizing the likelihood function. The likelihood function assigns a probability of observing a given set of data with a specific value assigned to the model parameters. Let $f(x;\gamma)$ be the parameterized function for which we wish to determine the maximum likelihood estimate of parameter γ. Assuming that the data is independent and identically distributed[6], the joint probability of observing the empirical data with a given parameter is

maximum likelihood

$$\Pr(X_1 = x_i, X_2 = x_i, ..., X_n = x_n) = \prod_{i=1}^{n} f(x_i;\gamma), \tag{2.8}$$

which is the equivalent to the likelihood function denoted by $\mathcal{L}(\gamma;x_i)$. For our specific case, the x_i denote the vertex degrees from the network (i.e., the observed data). The joint probability over every observation is then computed by using the product rule for independent events, $\Pr(x,y) = \Pr(x) \cdot \Pr(y)$. The likelihood function then represents the probability of observing this particular degree sequence assuming that it is sampled from a specific power-law distribution. By finding the value of the model parameter γ that maximizes the likelihood function, we can identify the specific power-law distribution that is mostly likely to have generated the observed data. For power-law distributions there is no exact closed form maximum likelihood solution, but an accurate approximation can be made for large sample sizes [CSN09]. The approximation for the scaling parameter is

$$\hat{\gamma} = \max_{\gamma} \mathcal{L}(\gamma) \simeq 1 + n \left[\sum_{i=1}^{n} \ln \frac{x_i}{x_{min} - \frac{1}{2}} \right]^{-1}. \tag{2.9}$$

This maximum likelihood estimator relies on knowing the value of $x_{\min}$ which, in general, needs to also be estimated. The selection of $x_{\min}$ is important because, if too low of a value is selected, then we will try to fit a power-law to data which is obviously not power-law distributed. In the case of choosing $x_{\min}$ to be too large, we throw away valuable data and decrease the accuracy (e.g., increase statistical error and bias from finite sample size) in estimating γ. The $x_{\min}$ can be found by performing a grid search of the parameter space

[6] The independence assumption means that $\Pr(x,y) = \Pr(x) \cdot \Pr(y \mid x) = \Pr(x) \cdot \Pr(y)$, and identically distributed means that all observations are stemming from the same power-law distribution.

to find the power law that best fits the observed data using a goodness-of-fit test [CSN09].

model fitness

The goodness-of-fit test is of crucial importance because the maximum likelihood estimate only indicates to us which power law best explains the data, but not if a power law is plausible model. It could very well be the case that the data could arise from a similarly shaped but fundamentally different kind of distribution (e.g., a distribution of the exponential family such as the Poisson distribution). The output of the goodness-of-fit test is a p value that quantifies the plausibility of the hypothesis. To test for the goodness-of-fit, we generate samples from the fitted power-law and compare these synthetically generated data to the empirical data. Formally, this is done using a Kolmogorov-Smirnov (KS) statistic. The KS statistic measures the distance between a set of sampled data and a hypothesized probability distribution. To perform this test, samples are drawn from the hypothesized model, in our case the fitted power-law, to generate an ensemble of synthetic data sets. For each synthetic data set, a power law is fitted using the maximum likelihood approach discussed earlier. The KS statistic is then computed between the synthetic data sets and their corresponding fitted power law. The p value is then computed as the fraction of KS statistics, from the synthetically generated set, that are larger than the KS statistic computed between the empirical data and its corresponding fitted power law. What this procedure tests is to what extend the disparity between the empirical data and fitted power law is explained by effects of finite sampling. If the disparity is significantly larger than what is expected from finite sample size effects, then we reject the hypothesis that the data could be plausibly explained by a power law distribution.

2.3.6 Modularity

Complex systems typically exhibit organized substructures, where elements of the system group together to serve a related function or capability. In the human body, cells make up the specialized substructures in the form of organs. A deep understanding of these substructures and their relationships to each other is critical to the scientific development of diagnostic tests for illnesses, effective treatment regimes, and preventative medicine to sustain good health and well-being. Similarly in complex networks, there is an interest in quantifying the extent to which a complex system is organized into modules and to identify the modules. Knowledge of the modules can help to understand functionally similar nodes in a network that may be especially well suited for a particular purpose. In software projects, the arrangement of software developers is often organized into a number of teams that persist over a period of time to help achieve efficient coordination as a result of familiarity and shared

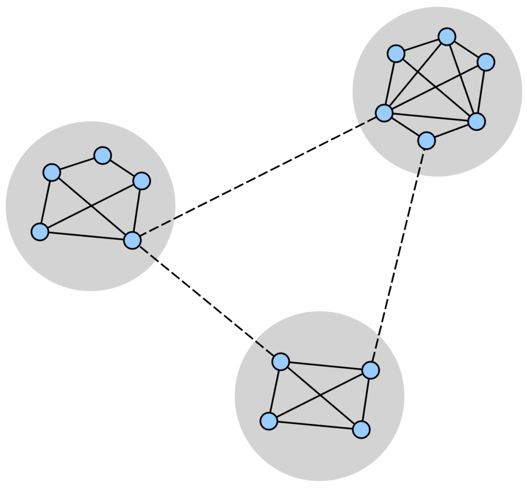


Figure 2.8: An example network that exhibits community structure. The key features of a community (gray circle) are (1) many within community edges (solid edges), and (2) comparatively few edges between distinct communities (dotted edges).

mental models (cf. Section 2.1). Since teaming as such an effective approach to managing coordination complexity that is inherent to software engineering, it is highly valuable to be able to identify and measure this organizational principle.

In a network, modules are synonymous with the terms *clusters* or *communities*, and are defined as subgraphs that exhibit the property of being internally densely connected (i.e., many edges between members of the subgraph) but externally sparsely connected (i.e., few edges between a member and non-member of the subgraph) (cf. Figure 2.8). To quantify the extent to which nodes of a network are organized into clusters, it is not necessarily required to identify the clusters. For example, the vertex clustering coefficient (cf. Section 2.3.3) captures notions of modularity at the vertex level. Since vertices of a cluster are internally densely connected, most of the neighbors of a vertex should be connected to vertices of the same cluster and therefore the neighbors will be densely connected. This is visible in Figure 2.8, where the nodes have a high clustering coefficient because most of each nodes neighbors are within the same cluster and within a single cluster the connection density is high so there is a high likelihood that the nodes neighbors are also connected. This type of graph topology results in the frequent occurrence of vertices with a higher clustering coefficient. This property is relative to a network exhibiting lower levels of modularity, where nodes do not associate into densely connected clusters, and so it is less likely for the neighbors a node to be connected. A nice property of using the clustering coefficient to quantify modularity is that it does not require partitioning of the network into clusters. That is beneficial because identifying an optimal partitioning is computationally difficult and often relies on probabilistic algorithms that maybe biased towards identifying clusters with certain properties (e.g., small clusters) [FB07].

In a complex network, the modules are often unknown a priori. *Community detection* or *graph clustering* algorithms are designed to identify the modules of a network based on topological properties. The goal of these algorithms is to partition the network into clusters that are maximally modular according to a modularity metric. Since the number of possible partitionings is large, even for moderately size graphs, a brute force search for the optimal partitioning is impractical. In general, the graph partitioning problem falls into the complexity category of NP-complete problems [Sch07]. The size of the solution space is large and grows quickly because a graph of size n has $\binom{n}{n/2}$ number of partitionings. For this reason, graph clustering algorithms necessarily rely on heuristics to generate close approximations to the optimal partitioning. Cluster quality metrics play a critical role in clustering algorithms by providing a means to measure if one partitioning is better than another. Many clustering algorithms work through iterative steps of moving nodes around to different clusters based on a heuristic. After each step, the modularity metric is evaluated. If an improvement is made the change in the previous step is maintained, otherwise the change is reversed. This is carried out until a convergence criterion is met or a maximum number of iterations is executed [Sch07].

cluster quality metrics

All clusters exists somewhere on a spectrum where one end point is a completely isolated subgraph and the opposite end point is a subgraph with no internal edges. Cluster quality metrics measure where a cluster or set of clusters exists on this spectrum. Arguably, the most common metric of cluster quality is *modularity* and is defined as

$$Q = \frac{1}{2M} \sum_{i \neq j} \left(A_{i,j} - \frac{k_i k_j}{2M} \right) \delta(c_i, c_j), \tag{2.10}$$

where $M = |E(G)|$, the number of edges in the graph. The c_i terms represents the cluster identifier that vertex i is a member of so that the δ-function zeros the summation term when the two vertices are assigned to different clusters. Inside the summation, $A_{i,j}$ is the network adjacency matrix and $k_i = \deg(i)$, the degree of vertex i. If connections are made uniformly at random, then the fraction of edges occurring between vertex i and j is $\frac{k_i k_j}{2M}$. Overall, the equation computes the difference between the observed fraction of within-cluster edge minus the fraction of edges expected to be within-cluster when edges occur between nodes at random. When a larger number of edges occur within clusters is much more frequent than what is expected from randomness, then modularity is high. Some deficiencies of modularity are that it is an overall evaluation of a given graph partitioning, that means nothing about the individual clusters is reflected. Modularity is also known to suffer from a resolution limit, that results in a bias toward large clusters [FB07].

Quality Metrics A number of other cluster quality metrics have been proposed in the literature. While all metrics suffer from one weakness or another, the *conductance* metric has shown to have good performance for a variety of cluster topologies and sizes [AGMZ11]. Additionally, conductance can reflect the quality of a single cluster. The conductance of a single cluster C in graph G is

$$\phi_G(C) := \frac{|\operatorname{cut}(C, G \setminus C)|}{\min\{\deg(C), \deg(G \setminus C)\}}, \tag{2.11}$$

where the cut operator computes the cut-set of a graph cut, that is the set of edges with exactly one end point in the cluster, and $G \setminus C$ is the vertex induced subgraph comprised of vertices in G but not in C. The sum of the vertex degrees in a subgraph is denoted by $\deg(C)$. If a cluster has zero within-cluster edges, then conductance is one. If all edges connected to vertices of the cluster are within-cluster edges, then conductance is zero. A potential disadvantage of conductance is that it does not included any kind of normalization for randomness, as is the case for the modularity metric. Therefore, conductance indicates the quality of the cluster, but not how likely it is for that level of conductance to occur with all edges of the graph are connected together uniformly at random.

One of the major challenges to overcome with using graph clustering algorithms is how to interpret the result. The problem stems from the fact that essentially all networks exhibit some level of clustering. Regardless of the network's topological properties or underlying organizational principles that give rise to the topology, clustering algorithms will in general produce an output containing one or more clusters. Still, the question of interest that motivates the use of clustering algorithms is whether there are relationships between elements of a common cluster that causes a bias towards the formation of within-cluster edges instead of edges crossing a cluster boundary. In a social network, groups of individuals may closely associate with each other as a result of shared interests [FLG00]. In the model of preferential attachment, the bias caused nodes to attach more frequently to nodes with a high degree. In the case of modularity, the bias causes a collection of vertices to associate more strongly with each other, and less strongly with vertices outside the cluster. Again, this is an example of a specific structure that is not explain by uniform randomness, but instead relies on a bias or preference in how edges form between vertices.

Essentially, the preference represents a departure from uniform randomness and implies the existence of self-organization.

Scale-free networks are particularly interesting because they are unlikely to arise from the ER random graph model. Structures that are a departure from uniform randomness are interesting because it is an indication of an organizational principle that can help to understand what forces influence the structure of a complex network. The scale-freeness property is a property of the degree distribution of a network, so the appropriate corresponding null model to represent a scenario with uniform randomness is the ER random graph. Similarly, to determine if a network partitioning (i.e., the output of a clustering algorithm) of an empirical network is significant in terms of identifying structures that are not a product of uniform randomness, a null model is also needed. An appropriate null model must be as closely related as possible to the empirical network, and only represent randomness in terms of the organizational principle under examination. For modularity, the organizational principle is a preference to attach to nodes that are members of a common cluster. On this basis, an ideal null model would then maintain all graph properties except exhibit uniform randomness with regard to this particular preference. However, the ER random graph model is an inappropriate null model for this purpose because it destroys not only preferences of modularity but also preferences in the degree distribution. A more appropriate null model for testing principles of modularity is the configuration model (cf. Section 2.3.4). Recall that the configuration model works by generating a network with a prescribed degree distribution, but then connects the nodes together uniformly at random. This model will maintain the number of nodes, number of edges, and the degree of each node, but does exhibit uniform randomness in terms of how collections of nodes are associated. Provided with a single empirical network, the configuration model can be used to generate an ensemble of synthetic networks that lack the organization principle leading to modularity. A clustering algorithm can then be applied to the synthetic networks to determine the level of clustering that is present in the graphs when no preference exists between elements of the network. This can then be compared to the empirical network clusters using cluster quality metrics to determine if the empirical network has clusters that are unlikely to occur from uniform randomness using standard hypothesis testing approaches.

2.3.7 Hierarchy

Scale freeness and modularity were discussed earlier as organizing principles that manifest as preferences for the nodes of a network to attach and associate with one another in ways that cannot be explained by uniform randomness.

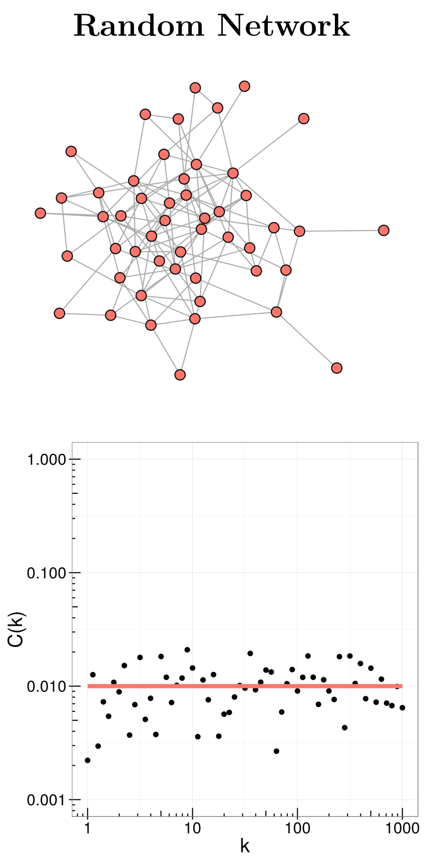
The concept of *hierarchy* is a third organization principle that brings these two concepts together by expressing how local groups are arranged relative to each other. In a hierarchical network, there exists stratification within the network that stems from cohesive groups being embedded within larger and less cohesive groups. This stratification is manifested as a relationship between the node clustering coefficient and the number of connections, that is, the node degree [RB03]. A comparison between the topology and key features that differentiate a hierarchical network from an ER random network is shown in Figure 2.9.In a hierarchical network, nodes with high degree and low clustering coefficient represent the top of the hierarchy and span multiple cluster. This property is visible in Figure 2.9, where the node located in the center of the hierarchical network is the highest degree node, but connected to nodes that are members of different cluster. At the bottom of the hierarchy are the low degree nodes that have a high clustering coefficient. This property is visible in Figure 2.9, where the low degree nodes are organized into clusters surround the central node. This topology is in contrary to the ER random network which does not exhibit these properties.

The key feature of a hierarchical network is a dependence between node degree and clustering coefficient and is described by $C(k) \propto k^{-\beta}$, where $c(k)$ is the clustering coefficient for a node with degree k [RB03]. In Figure 2.9, the scatter plot for an ER random network and hierarchical network is illustrated. Note that in the ER random network, there is no dependence between clustering coefficient and degree. To test for the presence of hierarchy, linear regression techniques can be used to solve for the optimal linear model satisfying the functional form $Y = \beta_0 + \beta_1 X$, where the clustering coefficient is the response variable denoted by Y and node degree is the predictor variable denoted by X. If the optimal linear model has a nonzero slope (i.e., $\beta_1 < 0$) and the slope parameter is statistically difference from zero, such that $p < 0.05$ where p is that probability that $\beta_1 = 0$, one can conclude that hierarchy is present.

Intuitively, this relationship implies that nodes of high degree tend to be connected to many different groups that are themselves loosely coupled to each other. What makes hierarchy particularly interesting is that it is not explained solely by preferential attachment and therefore indicates an entirely separate organizational principle [RB03]. Hierarchy is also indicative of the existence of an organizational structure that transcends the local network structure.

2.3.8 Network Evolution

So far, the discussion on networks and their structural properties have primarily been from a static view point. However, many of the phenomena that are represented as a network are actually dynamic and thus have a time-varying



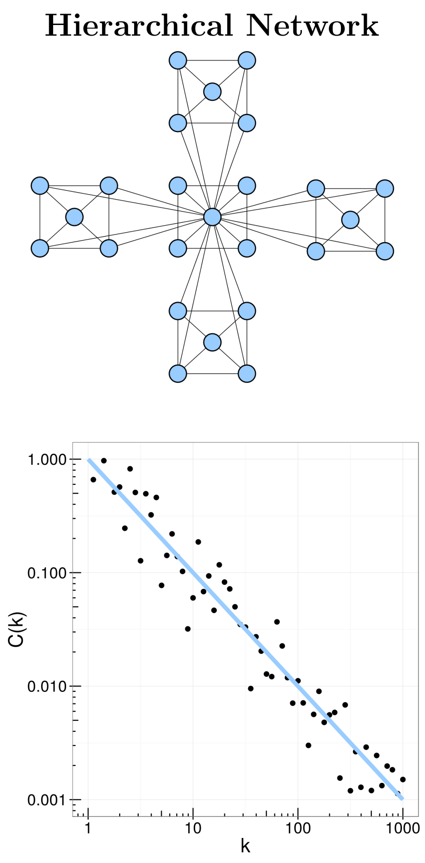


Figure 2.9: ER random network (left) and hierarchical network (right) with corresponding scatter plot of node degree k versus clustering coefficient $C(k)$. The hierarchical network topology deviates from randomness by having small cohesive clusters that are embedded withing larger and less cohesive clusters. The hierarchical topology manifests as a dependence between node degree and clustering coefficient, which is not present in ER random networks.

behavior. For exampled, the network structure of the World Wide Web (WWW), the Internet, and social networks, are all examples of networks that change with time. In software projects, the relationships between developers, the tasks they are assigned to, the people they work with, also change over time (cf. Section 2.1). If we wish to use network to model aspects of software development, then a dynamic perspective is also necessary. We first hinted at the dynamic properties of a network in Section 2.3.4 with the model of preferential attachment as one possible explanation for the generative process that leads to scale-free networks. By studying the time varying nature of networks, we are able to test hypothesis like preferential attachment. While static network properties explain important structural features, dynamic network properties explain by what process the structural features are generated.

In a static perspective, a network is representative of a single snapshot taking at some point in time. For the WWW network composed of Web pages and hyperlinks between Web pages, the static network reflects the state of the network at time t. By taking multiple snapshots at different points in time and placing them in time resolved sequence, a stream of graphs can be generated that represent the time varying structure of the network. The dynamic representation of graphs is simply a sequence of snapshots denoted by $G_1, G_2, \ldots, G_n$ where each G_t represents the graph snapshot at a time t. All of the static properties of graphs already discussed can then be computed for each graph in sequence to obtain the graph's evolution with respect to a specific graph property. For example, the degree of a vertex can be tracked through time by computing $\deg(v_{i,t})$, where $v_{i,t} \in V(G_t)$.

sequential data model

Adding the time dimension to study networks can lead to new insights, but there is also additional complexity in the modeling process. To explore the dynamics of networks, we need to understand the transitions that occur as the network evolves from G_t to G_{t+1}. For this purpose, sequential modeling techniques are particularly well suited and are designed to model the relationships of events that occur in sequence [Bis06]. The discrete time Markov chain is an example of a probabilistic sequential model that describes the probabilities of transitions between a set of states from time t_i to t_{i+1}. For example, let X_t be a random variable for the state of a node in a network. In this example, a node can be in one of three mutual exclusive states $X_i \in \{s_1, s_2, s_3\}$ defined to be the following.

- s_1: node is absent from graph
- s_2: node is present but isolated
- s_3: node is present and not isolated

For each node, a sequence represents the state that the node had at all points in time, $X_1 = x_1, X_2 = x_2, \ldots, X_t = x_t$. The Markov chain enables us to compute a probability that expresses the likelihood that a node in one state will transition to another state in the next iteration of the sequence. More formally, the probability we would like to compute is $\Pr(X_{t+1} = x | X_1 = x_1, X_2 = x_2, \ldots, X_t = x_t)$, that is the probability that the next state of the node is x, conditioned on all previous states. In many cases, the long range effect of state transitions is low, meaning that the most recent states contain the most information about the next state to occur in the sequence. The Markov assumption takes advantage of this common property by estimating the transition probability by $\Pr(X_{t+1} = x | X_t = x_t)$ [Bis06]. The Markov model can then compactly represent the transitions between states using only

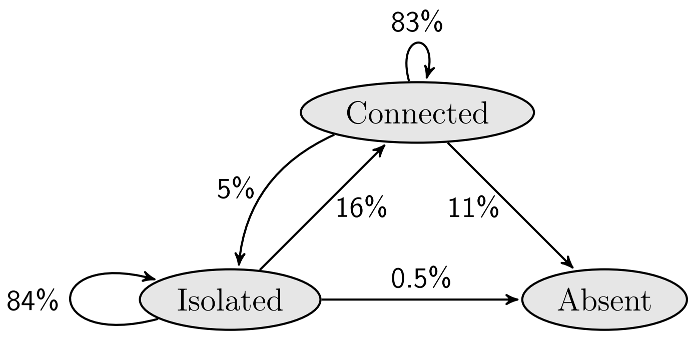


Figure 2.10: A Markov chain for nodes transitions between three states.

a small number of parameters, which makes them highly interpretable and computationally efficient to work with. In Figure 2.10, a Markov chain is shown with three states and the respective transition probabilities between states.

2.4 Developer Networks

The combination of software development being a labor intensive process and the widespread deployment of software repositories leads to the generation of data connecting individuals to the social and technical activities they participate in during software development. By identifying when two developers participate in a common activity, dyadic relationships between developers can be elicited from software repository data. The composition of all dyadic relationships involving all developers in a software project represents a network that captures the global topology of developer activity interdependence. Networks of this type are referred to as *developer networks* and can be generated by applying a variety of heuristics to a number of data sources with the overall goal of identify when developers are engaged in interdependent activities. A number of questionable assumptions and non-trivial leaps are necessary to abstract the real-world phenomenon of developer activities into a network representation, which may compromise the validity of the developer network in terms of reflecting accurate relationships. In the following sections, the general approach that is taken to abstract developer activities into a network structure from software repository data is described in detail to draw attention to where non-trivial leaps and simplifying assumptions are made and potential threats to validity exist. This is followed by specifics about how VCS and mailing-list data are used to construct developer networks.

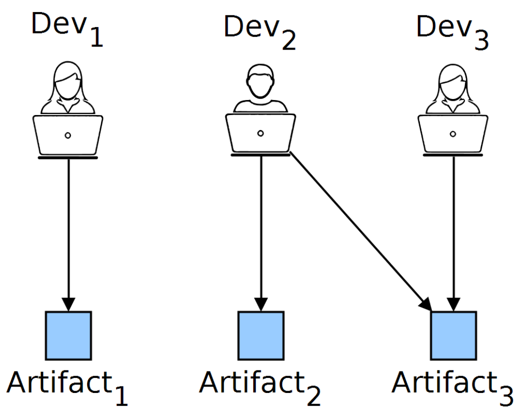


Figure 2.11: The information in software repositories provides explicit links between developers and the artifacts they generate or change.

2.4.1 General Framework

Despite the common claim that "networks are everywhere", we intend to show that a number of non-trivial operations are necessary to represent a real-world phenomenon as a network. We begin the process by reasoning directly from the relationships that are made explicit from the software repositories and then methodically step towards the final developer network. In Section 2.2, the version-control systems and mailing lists were introduced in terms of the data they store. In these repositories, links between people and the artifacts they touch are explicitly provided.[7] The data provided by software repositories is shown for three developers and three artifacts in Figure 2.11.

We see that what is depicted in Figure 2.11 is not a developer network with edges between developers but rather an affiliation or bipartite network with edges between mutually exclusive sets of nodes. One of these node sets is the collection of individuals that contribute to the project. In fact, even at this stage, it is important to recognize that we have made a strong claim that it is sensible and useful to represent the developers together as a collective set. This step can be rationalized by arguing that the collection of developers form one system and that the structure that emerges from the collective set of dyadic relationships between developers is of fundamental importance. The other node set that composes the bipartite network is that of artifacts (e.g., a subsystem, source-code file, source-code entity, etc.). For this set, a decision needs to be made about what constitutes an appropriate element of this set. Ideally, the contents that make up an artifact should be related as to suggest

[7]In this case, an artifact is essentially any tangible product of the work done by people (e.g., source-code files or e-mails).

that anyone touching a common artifact is likely to be performing related tasks or is involved in interdependent activities. Still, there is a whole spectrum of options that can be realized by either breaking down artifacts into smaller constituents, thereby moving towards finer-granularity, or aggregating several artifacts together, thereby moving toward coarser-granularity.

Formally, the bipartite graph is defined by two mutually exclusive sets, the developers $D = \{d_1, d_2, \ldots, d_n\}$ and the artifacts $A = \{a_1, a_2, \ldots, a_m\}$, and a set of edges E composed from one element of each set $E \subseteq D \times A$. The bipartite network in Figure 2.11 can be equivalently expressed in matrix form as

$$B = \begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \end{array} \begin{array}{c} \begin{array}{ccc} a_1 & a_2 & a_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \tag{2.12}$$

To make the leap to a network composed of edges between developers, again implies a set of non-trivial claims. The typical operation that is used to generate a developer network, though often not explicitly stated, is a one-mode projection of the bipartite graph defined by the following matrix operation

$$C_{\text{dev}} = B \times B^{\top} \tag{2.13}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \tag{2.14}$$

$$= \begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \end{array} \begin{array}{c} \begin{array}{ccc} d_1 & d_2 & d_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 1 & 1 \end{bmatrix} \end{array} \tag{2.15}$$

where C_{dev} is an adjacency matrix expressing relationships between developers based on their contributions to common artifacts. It should be immediately clear by this point that the one-mode projection of the original bipartite graph is less informative as it is in general a projection onto the subspace comprised of the developer set. The most obvious claim that we have made here is that two developers touching a common artifact implies something significant about those developers. A claim which is highly dependent on the nature of the artifact. We have also implicitly made the claim that all artifacts are essentially equivalent, though in reality, some artifacts may be connected to a large number of developers and are therefore indicative of a different kind of relationship. On the side of developers, we face a related simplification where we are now ignorant to whether a high degree developer is the result of touching a lot of artifacts or only a few artifacts that have many other contributors.

The nature of virtually any modeling process involves a set of assumptions. The main message here is that abstracting a real-world phenomenon into a network is non-trivial and requires assumptions that should be met with a healthy dose of criticism to ensure that the final representation maintains real-world validity. Without appreciation for the delicacy involved in constructing a developer network, any insight drawn from the network will have very limited value at best.

2.4.2 Types of Developer Networks

The basic information that is necessary to extract a developer network is traceability between developers and the artifacts they generate or contribute to. From this traceability, we are able to glean insight into the activities of developers. The nature of developer activities broadly fall into categories of social and technical and there are complementary developer networks that can also be broadly categorized according to this dichotomy.

Developer Coordination Network In developer networks of the technical variety, the relationship between developers that is expressed by edges in the network stems from activities that are technical in nature. Technical developer networks are primarily concerned with relationships that emerge from the development of source-code artifacts. By mining the version-control system, developers can be linked to all the code that they develop (cf. Section 2.2). The changes made to source code are group together according to a heuristic that defines the granularity of the artifact. In Figure 2.12, the information provided explicitly by the VCS and the resulting developer coordination networks are shown for two granularities.

Developer coordination networks can be constructed at various granularities. At the most coarse-grained side of the spectrum, source-code files are grouped together based on the project's top level directory structure [dSFD05; HL05; LRGH09; LRG$^+$04]. The assumption is that the top level directory reflects the subsystem-level decomposition of the software (cf. Section 2.2). The link between developers and the files they commit to can be easily obtained from the VCS and then the file-level contributions can be aggregated based the organization of the project's directory structure. While the technical difficulty involved in constructing this type of developer network is low, the

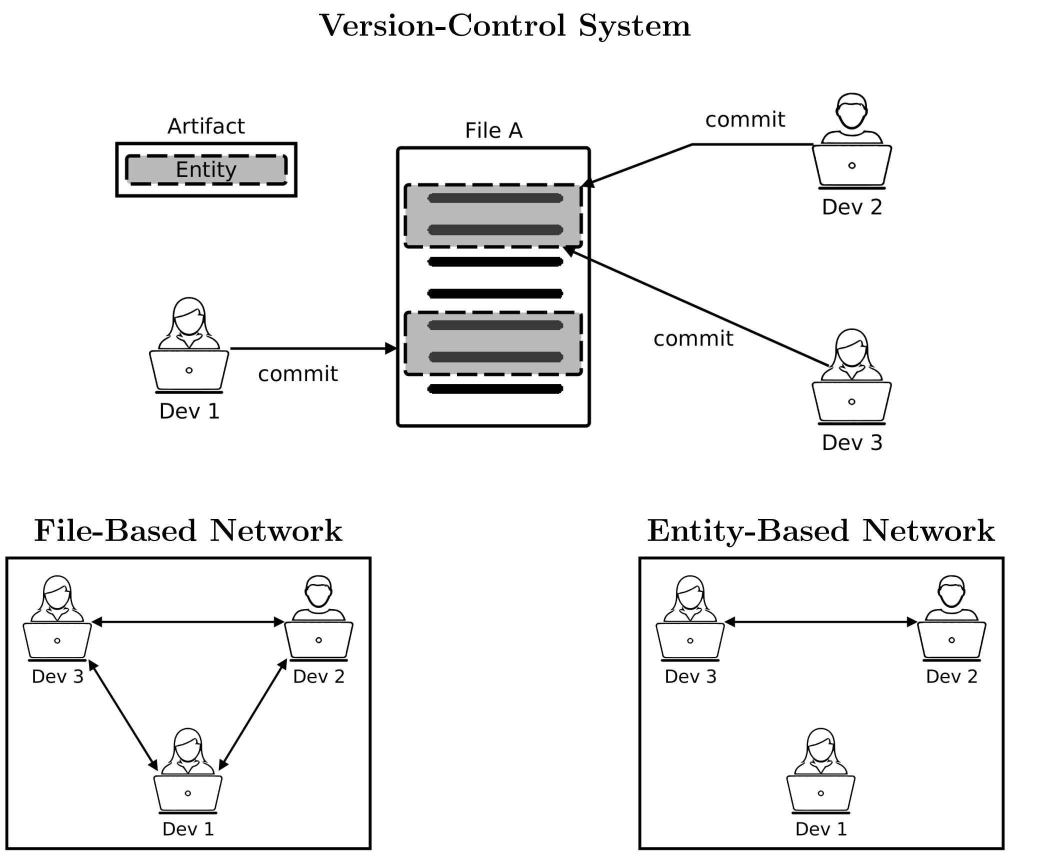


Figure 2.12: An illustration of the data provided by the version-control system (top), and the resulting developer coordination networks using a file-level granularity (bottom left) and entity-level granularity (bottom right).

real-world validity is questionable. While most software systems are divided into a number of subsystems, it is not clear whether that is always reflected by the projects top-level directory structure. It is also not clear whether two developers working on the same subsystem implies a meaningful relationship in terms of identifying whether the developers are engaged in interdependent activities. These subsystem-level developer networks exhibit heavy-tailed degree distributions and show evidence of being small-world networks, but in general very little is known about the topology, organizational principles, and real-world validity [HL05; LRGH09; LRG^{+}04]. In an effort to identify alternative network construction approaches that are more indicative of meaningful socio-technical relationships between developers, researchers have pursued the use of finer-grained artifacts. For example, instead of using top-level folders to group source-code, individual files are used to construct the developer network [JSS11;

MW09; MW11; PBD^{+}14; SMWH09]. In an abstract sense, a file can be viewed as a bundle of interdependent design decisions that results in code which is highly interdependent, but then more loosely related to the code in other files (cf. Section 2.2). Developers making changes to common files are thus expected to be engaged in interdependent activities. Developer networks of this kind exhibit real-world validity [MW11], but suffer from problems of over-connectedness that make it difficult to decompose the developer network into clusters without applying filters to remove edges [JSS11].

The edge weights of technical developer networks reflect the strength of the tie between two developers. This information is important because the activities of developers can exhibit varying degrees of interdependence. For example, an isolated change that is an atypical development task for a developer leads to edges in the developer network. In another case, multiple developers may work intensively on tasks thats persist over a long period of time leading to edges that are of a different nature. Ideally, the weighting function should be capable of differentiating between these scenarios. In the case of a binary weighting function, zero is assigned to an edge when two developers exhibit no relationship, and one is assigned when at least one commit has be made to a common artifact [LRGH09; LRG^{+}04]. This weighting function is simple, but is incapable of expressing relationship strengths. Another common weighting function uses the number of commits to common artifacts to assign weights [JSS11]. A third option is to count the lines of code that are contributed to common artifacts to assign weights [dSFD05].

Developer Communication Network In developer networks of the social variety, the relationship between developers that is expressed by edges in the network stems from activities that are social in nature. As developers work towards accumulating the necessary knowledge to complete their tasks and provide guidance to others, communication channels form between developers. An approximation to the complete set of communication channels between developers is expressed in the mailing lists of open-source projects (cf. Section 2.2). In Figure 2.13, a subset of data that is explicitly provided by the mailing list archives is shown along with the corresponding developer communication network. A developer network can be constructed by linking developers to the threads that they contribute, and then performing the one-mode project to generate the developer-developer edges [BPD^{+}08; PBD^{+}14]. The assumption is that developers that contribute to a common thread implies a personal interest in the discussion topic and awareness of the other individuals contributing to the thread. The major weakness and source of validity concerns in this approach is the content of the message is ignored. For this reason, it is unclear

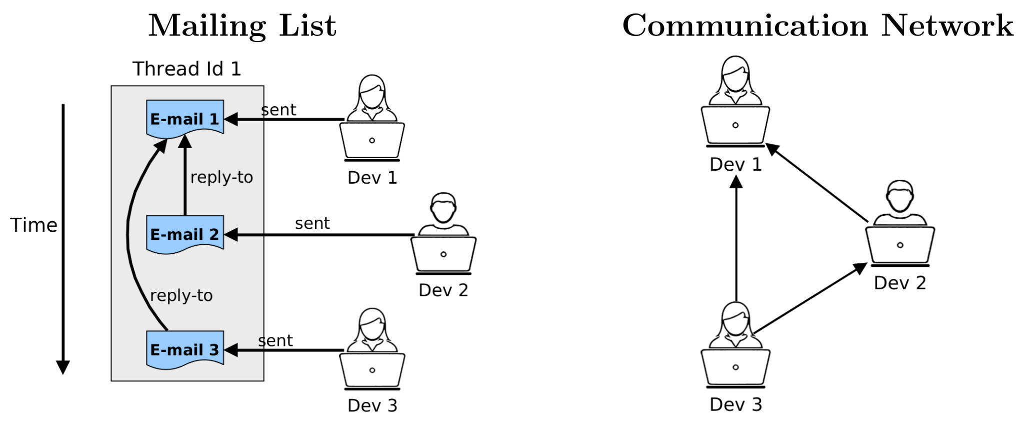


Figure 2.13: A depiction of the data provided by mailing list archives (left) is shown for three individuals that composed three e-mails in a single thread. The mailing list data provides links between individuals and the e-mails they have authored in addition to links between e-mails that expresses the Reply-to relationships. The corresponding developer communication network (right) stemming from activities in the mailing list is also shown.

whether a contribution to a thread has a positive sentiment (e.g., praise for good work) or a negative sentiment (e.g., criticism for inappropriate coding practices), or entirely off topic (e.g., vacation plans). While it is important to recognized that the developer network is certainly a simplification on the complexity of human communication, the network still contains insightful information. Social developer networks of several successful open-source projects have shown that mailing list communication between developers is correlated with activities in the source code [BPD$^+$08; XF14]. So, the social developer network also provide important information regarding where developers tend to have interdependencies in there tasks. The major difference between the social and technical developer networks is that the social relationships are more direct, and more strongly imply awareness of a coordination requirement, whereas with technical developer networks the relationships are more implicit and awareness of the interdependence is less strongly implied.

The edge weights are important in social developer networks for similar reasons as in technical developer networks. The intensity of communication between developers can vary significantly and familiarity with each other is expectedly higher the more frequent and temporally close the communication occurs. A binary weighting function, where zero weight indicates no communication, and one indicates at least one contribution to a common thread, is simplest but neglects the intensity of communication. A second and better option is to define the weighting function as the number of e-mails submitted to a common thread.

CHAPTER 3

Community Detection and Validation with Fine-grained Developer Networks

This chapter shares material with the ICSE'15 paper "From Developer Networks to Verified Communities: A Fine-Grained Approach" [JMA$^+$15].

Software development tasks (i.e. individual assignments of work) often exhibit some level of interdependence, which, to a large extent, stem from technical dependencies between source-code elements (e.g., data-flow dependencies). When interdependent tasks are assigned to distinct developers, these developers must coordinate their source-code modifications to avoid violated constraints that are imposed by the tasks' interdependence. Information about source-code changes (e.g., what code was changed and by whom) and source-code structure (e.g., which code is interdependent) provides important evidence of where coordination requirements exist between developers. From this evidence, developer networks can be constructed that represent the project's coordination structure. Although this rationale relies heavily on valid evidence of coordination requirements, and the validity of the resulting developer network, concerns regarding validity are, at this point in time, insufficiently addressed by the research community.

In Chapter 2, we introduced the notion of coordination requirements, the mechanisms that give rise to them, and how they present a significant threat to large-scale globally-distributed projects. As evidence of this threat, we discussed a high-profile software project where failure to adequately manage coordination requirements was the primary source of the project's failure. We also introduced teaming as a commonly used and highly effective technique for managing coordination requirements. Teaming is valuable because teams typically persist over a period of time, team members are familiar with one another and each others work assignments, and often have shared mental models making it easier for them to communicate effectively (cf. Section 2.1). While teams are known to be relatively effective at achieving adequate coordination, inter-team coordination is often the source of coordination breakdowns, because individuals are less familiar and may be unaware of each others actions. As an approach for modeling developer activity in software repositories, we introduced developer networks as an opportunity to gain perspective on the project's coordination structure (cf. Section 2.4). The theory of random graphs and techniques of network analysis are at our disposal to determine whether higher order structure is an inherent property of developer networks (cf. Section 2.3. In particular, community-detection algorithms are able to infer communities, which in many ways are similar to teams, that emerge from the arrangement of coordination requirements among developers (cf. Section 2.3.6)). In a practical sense, knowledge of the developers coordination structure, particularly, how they are arranged into teams, is enormously helpful to identify and bring awareness to potentially risky inter-team coordination requirements. Furthermore, knowledge about the developer organization helps to understand how well an organization is equipped to manage their coordination requirements. In essence, knowing which developers are members of which communities, and how those communities interact is of primary importance to managing coordination in software engineering.

Based on prior work from other researchers on developer networks (cf. Section 2.4), we identified that the most widely used approaches rely exclusively on a file-level or coarser (e.g., subsystems) granularity, to identify developer coordination requirements. At this level of granularity, any pair of developers that make modifications to the same file (or subsystem) are linked together. Since files can contain many hundred of lines of code and provide a multitude of independent functionality, modifications at the file level may be an unreliable source of evidence of coordination requirements by generating many false positives. In pursuit of more reliable evidence of coordination requirements, we propose two approaches. First, a fine-grained approach that relies on knowledge of source-code structure to identify lines of code that are more likely to be interdependent. Second, an approach that uses the VCS committer and author meta data to identify coordination between contributors that author source-code changes and the individuals that approve and integrate the changes (cf. Section 2.2). We also propose refinements on the network abstraction of developer coordination to include the temporal order of modifications by using edge directionality, and an edge weighting function to estimate the magnitude of coordination requirement. To investigate whether the finer-grained approach reveals a more realistic network, we compared the approaches and found that: (1) the fined-grained developer networks exhibit statistically significant community structures, that fail to appear in the file-based developer network counterpart and (2) the developer communities identified in the fine-grain network largely agree with developer perception and thus have real-world validity. To statistically evaluate developer communities, we propose a simulation technique that makes use of the configuration model for random graphs (cf. Section 2.3.4) to generate a null model, and an unbiased community quality metric to objectively measure community strength (cf. Section 2.3.6). To evaluate the developer networks with regard to real-world validity, we conducted a web-based survey of 53 developers from ten open-source projects. The results of this work provide the foundation on which to build tools and techniques for managing developer coordination in large-scale projects. Without the validity of the underlying developer network, no matter how sophisticated the tool or technique is, the impact generated by this line of work is limited and potentially damaging to the field of software engineering.

We have applied our approach to empirically study ten open-source projects, listed in Table 3.1. We chose the projects to demonstrate our methods' applicability to a wide range of projects, from a variety of domains, written in various programming languages, and ranging in size from tens of developers to thousands.

In this chapter, we make the following contributions:

- We define a general approach for *automatically* constructing developer networks based on source-code structure and commit information, obtained from a VCS, that is applicable to a wide variety of software projects.
- We study ten popular open-source projects and demonstrate that the state-of-the art method of constructing developer networks is unsuitable to identify fine-grained organizational features, while our approach is suitable.
- We demonstrate that committer–author information can be used to *automatically* construct developer networks with similar information as developer networks constructed using the manual certificate-of-origin reporting system for documenting the responsibility of code changes.
- We present an approach to statistically evaluate the existence of developer-network communities using state-of-the-art machine-learning algorithms and network-analysis techniques suitable for directed, weighted networks with overlapping communities.

- We validate our approach by questioning 53 open-source developers from ten different projects, and show that most developers agree that the networks accurately capture reality and the identified communities have real-world meaning.

3.1 Approach

We now present the details of our method for constructing fine-grained developer networks, based on information from the VCS and source-code structure, to identify when two developers are engaged in interdependent activities. Following the network construction, we introduce the statistical techniques we use to infer and verify developer communities.

3.1.1 Network Construction

We propose two methods for constructing developer networks, each of which captures different views on developer coordination. First, we introduce the function-based method that makes use of source-code structure to identify relationships between developers. Second, we introduce the committer–author-based method, which makes use of meta data from commits in the VCS to identify developer relationships. For background material on the fundamentals of developer networks and network construction, refer to Section 2.4.

3.1.1.1 Function-based Method

To construct a developer network, we use a heuristic for identifying when two developers are engaged in a coordinated effort. Coordination theory has established that the demand for coordination arises from inter-dependencies between the tasks carried out by a set of individuals [MC90]. Therefore, the validity of the heuristic is based on how accurately it can identify inter-dependent developer tasks. Previous research relied on file-based heuristic where developers were said to be coordinated when they made a contribution to a common file [JSS11; LRGH09; LRG$^+$04]. Advantages of using a file-based heuristic include ease of computation, programming-language independence, and suitability for heterogeneous documents (e.g., source-code and configuration files). The file-based method has certainly proved useful for studying global network properties (e.g., vertex degree distribution, average clustering coefficient, average shortest path length) [LRG$^+$04], however, we identified specific limitations of the file-based method that hinder community detection and justify a more fine-grained method (cf. Section 3.2).

Dyad Definition The activity of contributing code to a common file does not always demand or imply a coordinated effort because files often contain a multitude of different functionalities. In our *function-based* method, we use a more fine-grained heuristic based on code structure, where the event of two developers contributing code to a common function block is considered to be evidence of a coordination requirement between the developers.[1] An illustration of the function-based method is shown in Figure 3.1 for three developers working on two functions within a single file. The rationale is that code within a function block is inter-dependent as a result of accomplishing a relatively small task, which is the key principle of functional and procedural abstraction, and which indicates that the developers of that function are engaged in a coordinated effort. A finer-grained heuristic will invariably result in identifying a subset of the developer relationships implied by a coarser-grained heuristic. By using the function-based method, we consciously sacrifice some edges between developers in the corresponding developer network to gain the ability to detect developer communities and identify relationships between developers that are a more reliable indication of coordination needs. In Section 3.2.6, we empirically address this trade-off by testing whether the sacrificed edges are authentic with respect to capturing real-world coordination. In Section 3.2.4, we discuss how the file-based and function-based heuristics perform with respect to identifying developer communities.

Using the author information acquired from the VCS, together with structural information provided by Exuberant Ctags, we construct a weighted and directed developer network. Vertices of the network represent developers who authored the code, and edges are included between two developers only when both had made a contribution to a common function block.

To support numerous programming languages with our approach, we use the source-code indexing tool Exuberant Ctags to obtain the necessary structural information. Exuberant Ctags supports over 40 programming languages and is able to process thousands of files in seconds. It is necessarily based on heuristics for recognizing function blocks, but this is not problematic for our use case, as we discuss in Section 3.3.

Edge Direction Software development is achieved through incremental contributions, where one builds on previous work to introduce or improve features or functionality through commits, which are typically only a few lines of code [RKS12]. We capture this notion of incremental contributions by using the commits' timestamp for identifying the appropriate directions of the edges

[1]for example, the same function implemented in C or the same method or constructor implemented in Java

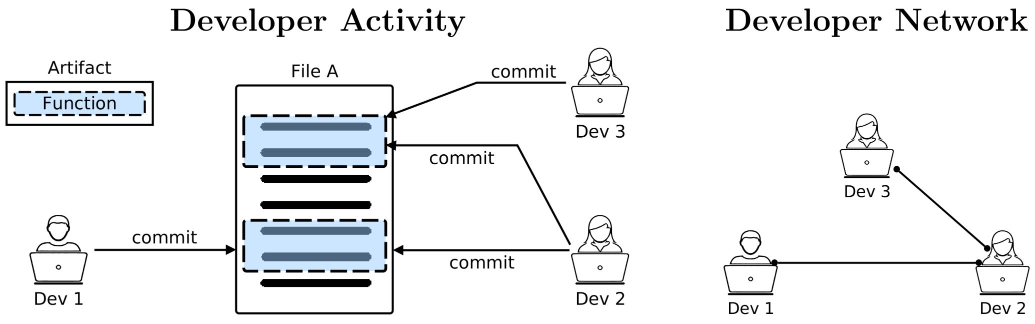


Figure 3.1: Three developers make modifications to two functions in a single file (left). The corresponding developer network from these modifications using the function-based heuristic is shown (right).

in the network. For example, developer A creates a new function without the need to collaborate closely with any other developer. At a later point, when that functionality is modified, developer B must understand and adhere to the constraints imposed by the remaining contribution of developer A. Thus, the dependency is unidirectional (developer A does not need to be aware of the contribution of developer B). By using directed edges, we enhance the graph by modeling an additional dimension of developer coordination, which is utilized by the community detection algorithm to more accurately identify communities.

Edge Weight Source-code modifications vary in size (e.g., in the number of lines of code) and vary in time (i.e. when the change is made). Depending on the modification's size and the time it's made, it will influence the work of other developers' modifications differently. For example, if two developers contribute many lines of code to implement a function, it is more likely that these developers are working on related tasks than if both developers only contributed one line of code. Time influences the nature of the relationship between developers because if one developer's code precedes the other, the first developer may never become aware of the later developer's changes. On the contrary, the later developer is constrained by the earlier developers modifications and must understand them to avoid violating assumptions imposed by existing code. To model these varying degrees of interaction between two developers from contributing to a common software artifact, we assign a weight to each developer relation in the network. To define the weighting function, we first need to introduce some notation. Let τ_{d1} denote the set of time indices for modifications made by developer $d1$ arranged in chronological order. For example, if developer $d1$ makes modifications at time $t = 1$ and $t = 8$ then $\tau_{d1} = \{1, 8\}$. Let $\mathrm{sloc}_{\mathrm{d1}}(t, f)$

represent the set of lines added or modified (neglecting white space additions) by developer $d1$ at time t to function f, then $|\,\mathrm{sloc}_{\mathrm{d1}}(t,f)|$ is the set cardinality. To represent the number of lines of code contributed to a function by a developer within a given time interval $[t_o, t_{o+\Delta}]$, we define

$$G(t_o, t_{o+\Delta}, f) = \sum_{i=t_o}^{t_{o+\Delta}} |\,\mathrm{sloc}_{\mathrm{d1}}(i,f)|. \tag{3.1}$$

We define the edge-weighting function $\omega_{d1,d2}(f)$ for developers $d1$ and $d2$ collaborating on function f according to:

$$\begin{aligned}\omega_{d1,d2}(f) = &\sum_{i=1}^{n} \Big[|\,\mathrm{sloc}_{\mathrm{d2}}(\tau_{d2,i},f)| + G(\tau_{d2,i},\tau_{d2,i-1},f)\Big] \\ &\cdot \delta(G(\tau_{d2,i},\tau_{d2,i-1},f)),\end{aligned} \tag{3.2}$$

where $\delta(x)$ is given by,

$$\delta(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

Equation 3.2 reflects the expected degree of coordination requirement between developers as a function of both temporal location and amount of contributed code made through successive changes. The inner summation captures the consecutive nature of one commit building upon the development work of all previous commits. The outer summation sums over all commits made by developer $d2$ to function f. Equation 3.2 considers directionality of edges, therefore $\omega_{d1,d2}(f) \neq \omega_{d2,d1}(f)$ in general. Finally, the total weight between $d1$ and $d2$ is

$$w_{d1,d2} = \sum_{f\in F} w_{d1,d2}(f), \tag{3.4}$$

where F is the set of all functions. An example calculation for the edge weights based on source-code changes is provided in Figure 3.2.

3.1.1.2 Committer–Author-based Method

Our second method is inspired by earlier work that used sign-off tags on commit messages to build developer networks [BRB+09]. In this method, tags are used to identify relationships between all people that contributed to a common commit, including authors, reviewers, and testers. An example of a commit that contains various tags is shown in Figure 3.3. In this particular commit,

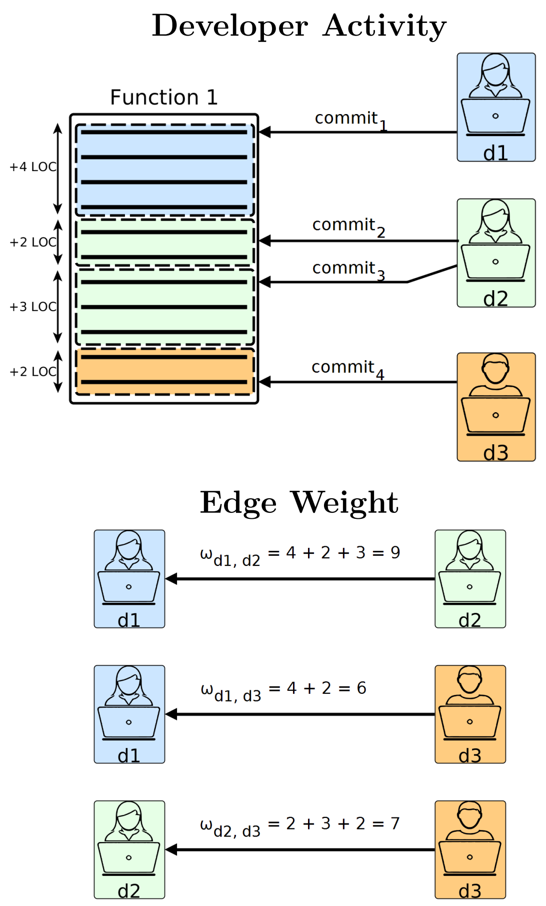


Figure 3.2: Three developers modify a single function block with a series of commits indexed according to time (top). The edge weight calculation is shown for the three developers that result from their modifications (bottom).

we are aware of three different individual that are involved in the change, that included reporting an issue (Andy), implementation of a change to address the issue (Michael), testing (Andy), and committing the change to the repository (Linus). A tag-based network contains important information about the software-development process, workflow, and developers with related interests and knowledge [BRB$^+$09]. Sign-off tags are self-reported acknowledgments of participation on a commit, therefore the tag-based networks undoubtedly capture real-world coordination. Unfortunately, only a small number of projects currently use the tag convention.

Our solution for projects that lack the tagging convention is to use the distinct author-and-committer information captured by Git to construct the network. For every commit, we place a unidirectional edge pointing from the

```
Meta data
commit: 2ac46030331e2952a56c...
Author: Michael S. Tsirkin <mst@redhat.com>
Committer: Linus Torvalds <torvalds@linux.org>
Date: Sun Nov 15 15:11:00 2015 +0200

Commit Message
virtio-net: Stop doing DMA from stack

Once virtio starts using the DMA API, we won't be able to
safely DMA from the stack. virtio-net does a couple of config
DMA requests...

Reported-by: Andy Lutomirski <luto@kernel.org>
Signed-off-by: Michael S. Tsirkin <mst@redhat.com>
Signed-off-by: Linus Torvalds <torvalds@linux.org>
Tested-by: Andy Lutomirski <luto@kernel.org>
```

Figure 3.3: Example of the information that is contained in a commit, including "Sign-off" tags documenting the trail of individuals responsible for reporting, testing, and fixing the issue.

committer to the author. In reference to the commit shown in Figure 3.3, an edge would be placed between the author Michael S. Tsirkin, pointing toward the committer Linus Torvalds. The direction is important, since relationships of this type are not necessarily reciprocal and the role of author and role of committer is fundamentally different. A weight for each edge is the sum of the number of commits with a common author-and-committer pair. Formally, we define the weighting function as

$$\omega_{d_1,d_2} = \sum_{i=1}^{n} \text{committer}(d_1, c_i) \cdot \text{author}(d_2, c_i), \tag{3.5}$$

where the operator committer(d_j, c_i) is equal to 1 when developer d_j is the committer of commit c_i, and 0 otherwise. Similarly, the operator author(d_j, c_i) is equal to 1 when developer d_j is the author of commit c_i, and 0 otherwise.

Since tag-based networks represent factual real-world collaborative structures, we use them (if available) to validate the structures of the automatically constructed committer–author-based networks. In Section 3.2.5, we show that the committer–author network of Linux is able to capture the same information as the corresponding tag-based network.

3.1.2 Network Analysis

We now discuss how we use network-analysis and statistical methods to infer *statistically significant* communities. Statistical significance is an important factor because all networks, even ones with nodes connected uniformly at random, exhibit some level of community structure. In this case, statistical significance allows us to determine if the developer organization is a consequence of non-random organizational principles or just random behavior. An example of a non-random organizational principle may be that of teaming, where developer associate strongly with a group of individuals that have share responsibilities, interests, or skills. For an introduction to the mathematical notation of graphs that is used in the following and for an introduction to network analysis basics, refer to Section 2.3. Additionally, in Section 3.2.6, we validate the community's *real-world significance* by surveying the developers that participate in the detected communities.

3.1.2.1 Community Detection

In the most fundamental sense, community-detection algorithms allow us to decompose an arbitrary network into communities without a-priori information about the communities (e.g., number of communities, their size, etc.) [BHKL06; EM12; LRRF11]. For further details on community detection principles, refer to Section 2.3.6. One important technical limitation of most community-detection algorithms is the inability to handle weighted and directed graphs, and many more are unable to identify overlapping communities. The implication of non-overlapping communities is that community membership is unique (i.e. each developer is a member of exactly one community). In the case of developer networks, we expect important developers to lie at the boundary between two or more communities. If overlapping communities are not permitted, a developer will be incorrectly forced to exist in one community.

For community detection, we use the order statistics local optimization method (OSLOM), which has not been done before on developer networks. The OSLOM is an improvement over other community detection methods because the optimality criterion is based on a fitness measure that is parameterized using notions of the statistical significance of a cluster. Most other community detection techniques rely solely on optimizing modularity (cf. Section 2.3.6), but these approaches can result in identifying clusters that are still statistically likely to occur in a random network [LRRF11]. Instead, OSLOM defines the fitness of each cluster based on the probability of finding the given cluster in a class of networks without any community structure. Nodes that lessen the statistical significance of a cluster are then pruned or added to alternative clusters. The result is that if a random graph containing no community structure is feed into OSLOM, the output will contain few clusters or none at all. An additional feature of the OSLOM approach is that it is one of the few methods that is able to handle weighted and directed networks and to identify overlapping communities [LRRF11]. As is shown in Figure 3.4, it is not uncommon that a node exhibits similar levels of connectivity to two distinct communities. Without allowing for overlapping communities, the red node in Figure 3.4 is forced to exist in a single community using an arbitration rule. By forcing the node's membership into a single community, any relationship that node had with members of other communities is completely lost and neglected.

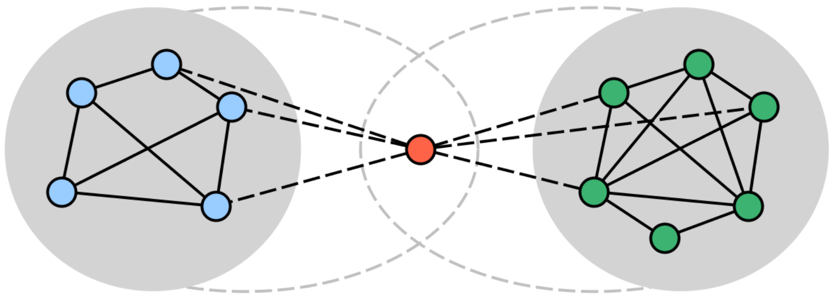


Figure 3.4: Nodes belonging to a blue community and green community are shown. The red node exists at the boundary of the communities by having similar levels of connectivity to both communities. Without allowing for overlapping communities, the red node is forced to be a member of a single community. The dotted gray boundary demonstrates how each community could reasonably expand its boundary to include the red node.

During the development of our approach, we experimented with several other community-detection algorithms and experienced generally poor performance from basic techniques, such as random-walk or eigenvector based methods [KVV04]. A statistical-mechanics approach using spin-glasses had comparable performance to OSLOM, but it does not produce overlapping communities [EM12].

3.1.2.2 Community Verification

The validity and interpretation of the identified communities is often unclear because community-detection techniques inherently rely on principles of unsupervised learning. Alternatively, a ground-truth naturally exists in the case of supervised learning approaches that can be conveniently used to establish a basis for model validation by means of a technique known as cross validation [Bis06]. In the unsupervised learning approach, an important step that is often neglected is to validate the output of community detection algorithms by determining whether the identified communities are meaningful [LRRF11]. A meaningful community, in this sense, is one that indicates an organized process or an organization principle that binds members of a community together. In contrast, a community that arises simply by random chance occurrence is not meaningful and probably elicits no interpretable insight.

We assess the validity of the observed communities by computing the probability of observing the community in an equivalent class of null-model graphs that lack a community structure. The experimental setup is such that our null hypothesis is that developer networks contain no meaningful community structure. Therefore, the null model reflects the developer network under a scenario where developers' activities are as usual (i.e. developers still contribute code at their usual rate and coordinate with the same number of people), except the developers do not associate (i.e. have coordination requirements) with a particular group of individuals. For the null model to be valid, it must represent a typical development process (i.e. the same number of developers, the same commit rate, etc.), and only differ in the variable that we want to investigate, that is the variable of community structure. Since we do not know a priori which developer networks contain meaningful communities, after all that is exactly what we wish to determine, we make use of a simulation procedure to construct the null model developer networks from the empirical (i.e. observed) developer networks. Then we are able to compare the strength of the communities in the null model networks and the empirical networks. If the community strength in the observed and null model networks is equivalent, then the observed communities must not be meaningful because they can arise from a random process.

To generate the null model, we use a standard approach called the *configuration model for random graphs* (described in Section 2.3.4), where nodes are joined uniformly at random with the constraint that the degree distribution is identical to the input graph [GMZ03]. Since the degree distribution is maintained, the null model generation procedure does not alter the activity level of a developer (i.e. the amount of code contributed), or the number of people a developer has coordination requirements with. If it is possible to detect a statistically significant difference, in terms of community strength, between the null model and observed developer communities, we can conclude that it is improbable that the topological structure of the observed developer network arose from a uniformly random process and is more likely explained by an organized process, such as a coordinated development effort and teaming.

The structure of a community, or set of communities, is measured according to *community-quality metrics*, of which several have been proposed in the literature [AGMZ11]. In basic terms, nodes that are members of the same community are densely connected and comparatively loosely connected to nodes that are not members of the same community. A variety of communities are realized by varying the amount of internal density and external sparsity of edges. Community quality metrics are used to measure the extent to which a subgraph adheres to the theoretical properties of a community. For an introduction to network modularity and community-quality metrics, refer to Section 2.3.6. In our approach, we avoid the commonly used modularity metric in favor of *conductance* for four reasons [LLM10]. First, modularity is known to suffer from a "resolution limit", meaning it is unable to reliably measure small communities [FB07]. Second, modularity is often the optimization criterion used by community-detection algorithms. By using conductance, we avoid topological-structure bias introduced by the optimality criterion imposed by the community-detection algorithms. Third, conductance allows us to characterize an individual community, whereas modularity is a global metric that considers all identified communities and does not have a meaningful interpretation for a single community [KVV04]. Fourth, modularity is known to increase with the number of communities and nodes, making it inappropriate to compare projects of different size [For10]. Although all known community-quality metrics suffer from some type of bias, conductance has been shown to exhibit reliable behavior for a wide range of cases [For10].

Formally, conductance $\phi \in [0, 1]$ of a community C in graph G, such that $V(C) \subseteq V(G)$, is defined as:

$$\phi_G(C) := \frac{|\operatorname{cut}(C, G \setminus C)|}{\min\{\deg(C), \deg(G \setminus C)\}}\,, \tag{3.6}$$

where cut is the cut-set of a graph cut, and deg is the total degree of a graph [AGMZ11]. Intuitively, ϕ is the probability that a random edge leaves the vertex set that composes the community. An isolated community, with no edges leaving the community-vertex set, has zero conductance. Conversely, a community with every edge leaving the community-vertex set has a conductance of one. It is important to recognize that ϕ is a function of both intra-cluster and inter-cluster edges.

To discriminate between identifying statistically significant communities and purely random topological features of the network, we employ a stochastic simulation. Given a developer network G with $N \equiv |V(G)|$ vertices (developers) and E edges (connections between developers), we apply a community-detection algorithm to identify a set of communities $\mathcal{C} = \{C_1, C_2, \ldots, C_i\}$ where $V(C_i) \subseteq V(G)\ \forall i$. Mean conductance over all communities is given by

$$q_G(\mathcal{C}) = \frac{\sum_{C \in \mathcal{C}} \phi_G(C)}{|\mathcal{C}|}. \tag{3.7}$$

Using this input data, we generate a null model that represents an equivalent developer network but with disorganized activity. To generate the null model, we randomize the original network according to the configuration model using a graph-rewiring technique, with which the pairs of edges are selected uniformly at random and the end points swapped, such that an edge pair (u, v) and (s, t) is rewired to (u, t) and (s, v) [GMZ03]. The rewiring procedure maintains the amount of connectivity (i.e. number of edges) for each developer, but destroys the preference to attach to a particular group of developers. The reason for destroying this preference is that the resulting network will not contain community structure that is a product of an organized process. Any communities that remain will exhibit properties that reflect random associations between developers. The rewiring procedure is executed m times[2] to generate a set $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$ of rewired graphs with $V(R_i) = V(G)\ \forall i$. This set of rewired graphs represent a family of networks that all contain various amounts of community structure, but all of which are a product of a random process. From this distribution of networks, we gain insight into which kinds of communities are likely and unlikely to occur in a developer network where there is no preference (i.e. organizational principle) for developers to associate with particular groups.

It is important to recognize that this randomization process is highly constrained compared to the ER random graph model described in Section 2.3.4 because it maintains the degree distribution, edge weights, and edge direction. The degree distribution, which represents the amount of participation by each developer, is given by $P_C(k) = |\{c \in C|\deg(c) = k\}|/|N|$. The rewiring procedure is intentionally designed to maintain the original degree distribution, that is,

$$P_{R_i}(k) = P_G(k) \quad \forall R_i \in \mathcal{R}\,. \tag{3.8}$$

For each rewired graph R_i, we calculate the mean conductance $q_{R_i}(\mathcal{C})$ and define the probability distribution as

$$P_{\mathcal{R}}(Q) = \frac{|\{i \in [1, |\mathcal{R}|] \mid q_{R_i}(\mathcal{C}) = Q\}|}{|\mathcal{R}|}\,. \tag{3.9}$$

With standard hypothesis testing, we can then evaluate whether the coordination structure is statistically significant. We check whether it is possible that

[2]We ensured that our choice $m = 3000$ was sufficiently large by checking the convergence of all derived results.

the observed graph could be described by the generated null model with the equivalent degree distribution as the observed graph; if this is not the case, we conclude that our observation is not described by a uniformly random process.

More precisely, the null hypothesis H_0 that the observed mean conductance $q_G(\mathcal{C})$ is described by the conductance distribution of the rewired (null model) graphs with a nonvanishing probability is given by,

$$H_0 : P_{\mathcal{R}}(Q = q_G(\mathcal{C})) > \epsilon \,, \tag{3.10}$$

with the alternative hypothesis given by $H_1 : P_{\mathcal{R}}(\cdot) \leq \epsilon$.

We use a one-sample t test to evaluate the hypothesis with the standard significance level of 0.05. Since the t test is robust against the deviation from a normal distribution with large sample sizes (i.e., larger than 30), we do not need to check our data for a normal distribution. We present the results of the statistical test for all subject projects next.

3.2 Evaluation & Results

We now present our hypotheses and findings on the network properties of developer networks we constructed for ten open-source projects. To address our hypotheses, we compare developer networks constructed using the prevalent file-based method (cf. Section 2.4) and the more fine-grained methods we propose (cf. Section 3.1.1.1). In Section 3.2.6, we present the results of a developer survey to address the validity of our approach with respect to capturing real-world coordination. Our overall goal is to determine whether the methods generate network representations of developers' activities that exhibit higher-order structures, in terms of developer communities, and exhibit real world validity. With a valid network representation, we are able to derive value from knowing which developers are likely working with each other on common or related tasks. In addition, a valid and significant community structure is indicative of mechanisms that give rise to a pattern and is suggestive of an organized development process.

3.2.1 Hypotheses

In order to derive value and utility from developer networks, previous work has largely assumed that the networks are an accurate representation of developer coordination and this strong assumption has been challenged in only a limited scope [MW11]. We challenge this fundamental assumption about developer networks by investigating the local topological features that should be present if the network is indeed an authentic representation of developer coordination.

Though other views are possible, we then validate that this particular view on coordination aligns with developers' perceptions (cf. Section 3.2.6).

Software development is an organized process and, if a developer network faithfully captures real-world developer coordination, it should also exhibit an organized structure.

H1—*Developer networks exhibit identifiable communities that significantly exceed the magnitude of organization that is expected from an equivalent unorganized process.*

By an equivalent unorganized process, we mean a situation that is equivalent to the original process except that developers' contributions to the software system are randomly distributed across various system components, showing no particular organized responsibility toward a particular aspect of the system.

The standard method of constructing developer networks relies on file-level information to identify collaborating developers. We show that this method is insufficient for identifying the latent community structure as a result of over-connecting developers in the network. Dense networks are known to hinder community-detection algorithms [BGW03]; furthermore, prior work has shown this problem arises for file-based developer networks [JSS11].

H2—*Developer networks constructed using the standard file-based method fail to identify statistically meaningful communities, whereas a more fine-grained function-based method is able to identify statistically meaningful communities.*

The manual process of tagging a commit is an intentional acknowledgment of one's participation in a commit. Each developer only tags a commit once they have made a contribution to the code. Therefore, a developer network constructed on the basis of commit tags can be regarded as a faithful representation of real-world coordination. To evaluate the validity of the committer–author-based method, we quantify congruence between the ground truth tag-based network and our automatically-constructed committer–author network.

H3—*Tag-based developer networks constructed from the manual process of tagging commits are highly congruent with automatically determined committer–author-based networks.*

3.2.2 Subject Projects

We selected ten open-source projects, listed in Table 3.1, to evaluate the methods we proposed; the projects vary by the following dimensions: (a) size (lines of source code from 50 KLOC to over 16 MLOC, number of developers from 15 to 500), (b) age (days since first commit), (c) technology (programming language, libraries used), (d) application domain (operating system, development,

Table 3.1: Overview of subject projects for a 90-day development window.

Project	Developers	MLOC	Lang	Domain
Linux	580	16	C	OS
Chromium	500	6.5	C, C++	User
Firefox	400	9.3	C++, JS	User
GCC	70	6.2	C, C++	Devel
QEMU	50	0.78	C	OS
PHP	50	2.2	PHP, C	Devel
Joomla	30	1.3	PHP, JS	Devel
Perl	30	4.5	Perl, C	Devel
Apache http	15	2.2	C	Server
jQuery	15	0.05	JS	Library

productivity, etc.), (e) development process employed, and (f) VCS used (Git, Subversion).

For each project, we analyze the VCS for a 90 day window starting in the second quarter of 2014. While window size certainly influences the resulting network, the impact of enlarging the window beyond 3 months is marginal [MW11].

3.2.3 Existence of Statistically Significant Communities

To test hypothesis H1, we used our function-based method to construct developer networks for all subject projects. We expected statistically significant communities to exist as a result of an organized software-development process in, at least, some of the subject projects, and we now evaluate whether our method is able to identify the communities using the community-verification procedure described in Section 3.1.2.2.

As an example, taken from QEMU, of the primary result, Figure 3.5 shows the histogram and kernel density estimation (Gaussian kernel) for the conductance samples drawn from the set of 3000 rewired networks. The majority of the probability mass is located between a conductance value of 0.5 and 0.625. This range of values represents what is expected from a network with no meaningful community structure. Recall that smaller conductance indicates a stronger community, and equivalently, higher levels of organization. The data indicates that observing a community with a conductance smaller than 0.5 is extremely unlikely to occur in the null model. In Figure 3.5, the observed conductance (i.e. from the non-rewired network), is represented as a solid point located at 0.38. Notice that the this point lies significantly left of (stronger

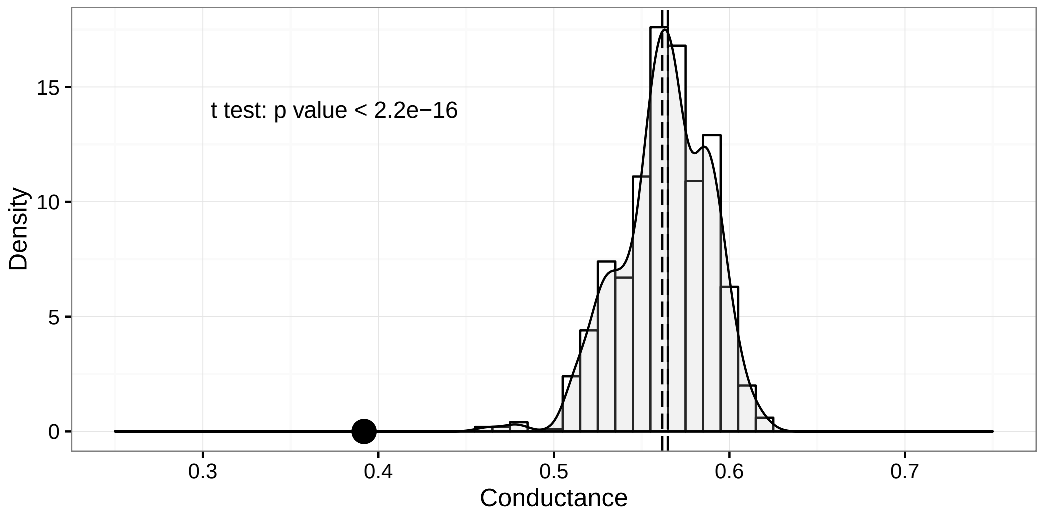


Figure 3.5: Community significance test on the observed mean community conductance (black dot) against the distribution of mean community conductance for 3000 rewired graphs for QEMU development from 14.2.2014 to 14.5.2014. Vertical lines represent the 95% confidence intervals.

community) than any of the samples from the rewired networks. This clearly shows the separation between the observed developer-network conductance of QEMU and the conductance distribution for the unorganized (rewired) network. The small p value of the t test indicates that the observed communities are statistically significant. Table 3.2 summarizes the results for all the subject projects: The function-based method identifies strong communities in several of the subject projects and for all, a statistically significant difference between the observed and rewired networks.

In conclusion, we reject the null hypothesis that the observed developer networks exhibit communities with properties that could plausibly arise from an unorganized process. Thus, we *accept H1*.

3.2.4 Comparison of File-Based and Function-based Methods

To test hypothesis H2, we performed a comparison between the file-based method and the function-based method for constructing developer networks (cf. Section 3.1.1.1). The comparison draws attention to limitations of the file-based method that manifest as the inability to identify statistically significant communities. In particular, we evaluated the mean community conductance and mean community density for two revisions of each subject project. Graph

Table 3.2: Comparison of community conductance for the original (observed) and randomized (rewired) networks.

Project	Observed conductance	Rewired conductance
Linux	0.05	0.88
Chromium	0.20	0.74
Firefox	0.11	0.79
GCC	0.01	0.48
QEMU	0.39	0.56
PHP	0.15	0.80
Joomla	0.57	0.84
Perl	0.49	0.66
Apache http	0.27	0.80
jQuery	0.49	0.75

density is a measure of graph connectedness, where a complete graph has density 1, and a graph with no edges has density 0. Figure 3.6 shows a scatter plot of mean community conductance versus mean community density, in which each point represents a three month project revision. We see an approximate but distinct separation between the file-based and function-based networks. Communities identified in the function-based network are both internally dense and exhibit low conductance (i.e., strong community structure). In contrast, the file-based communities are dense, which we would expect because of the overall high density of the network, but exhibit high conductance (i.e., weak community structure). From this result, we can conclude that the edges that are neglected by the function-based method are the ones which cross community boundaries. For this reason, we see the function-based and file-based communities exhibit similar levels of internal density, but the conductance in the function-based communities is lower. In Section 3.2.6, we address the relevance and validity of the file-based edges that are crossing a community boundary and ignored by the function-based method. In summary, this result demonstrates that the finer granularity of the function-based method enables the discovery of statistically significant communities, but is not excessively fine such that it destroys the connectedness of the graph.

The probability density plot shown in Figure 3.7 further illustrates the significant difference between the function-based and file-based network communities. There is a clear separation between the distributions where the function-based method identifies significantly stronger communities compared to the file-based method. We performed a paired t test to evaluate whether the difference in

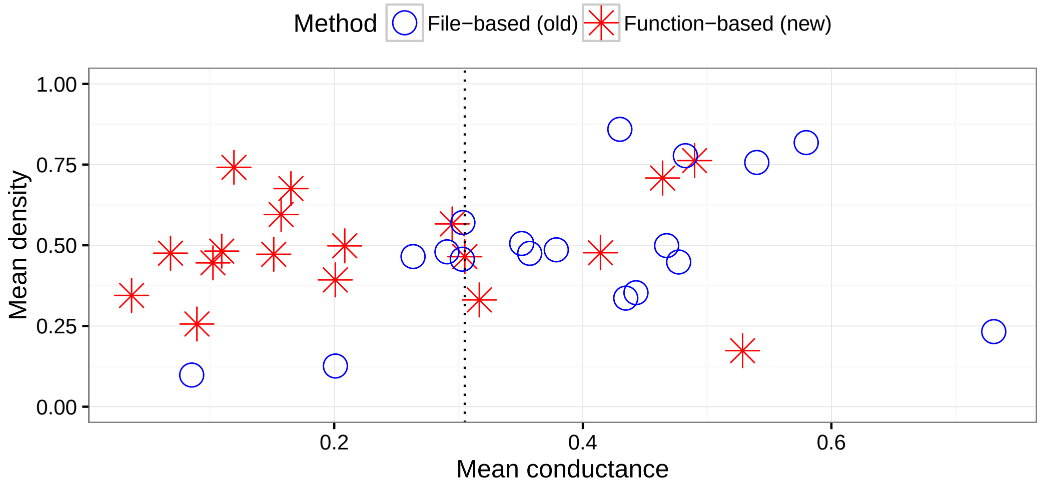


Figure 3.6: Scatter plot of projects analyzed using both file-based and function-based methods for two different revisions. A clustering by crosses (left) and circles (right) is visible; the function-based method is able to resolve more significant communities without compromising density.

the distributions is statistically significant. Before performing the t test, we checked the distributions for normality using a Shapiro-Wilk test. It produced p values of 0.99 and 0.075 for the file-based and function-based distributions, respectively. The following t test generated a p value of 1.29×10^{-5}. Thus, we can confidently reject the null hypothesis that the difference between the two measurements has a zero mean value.

To draw further attention and insight to the differences between file-based and function-based communities, we visualize the developer networks of QEMU in Figure 3.8. It illustrates the inability of the file-based method to identify statistically meaningful communities with the example of the QEMU developer network. Each bounding box represents a single community of developers. The border color of each box uniquely identifies each community, and pie charts are used to represent each developer's relative participation in a community. A box's background color is used to represent the significance of each community, calculated according to the conductance distribution (cf. Section 3.1.2.2). Green represents a significant (strong) community and yellow represents an insignificant (weak) community. Intra-community edges are shown in black, and inter-community edges are shown in red. The edge thickness represents the

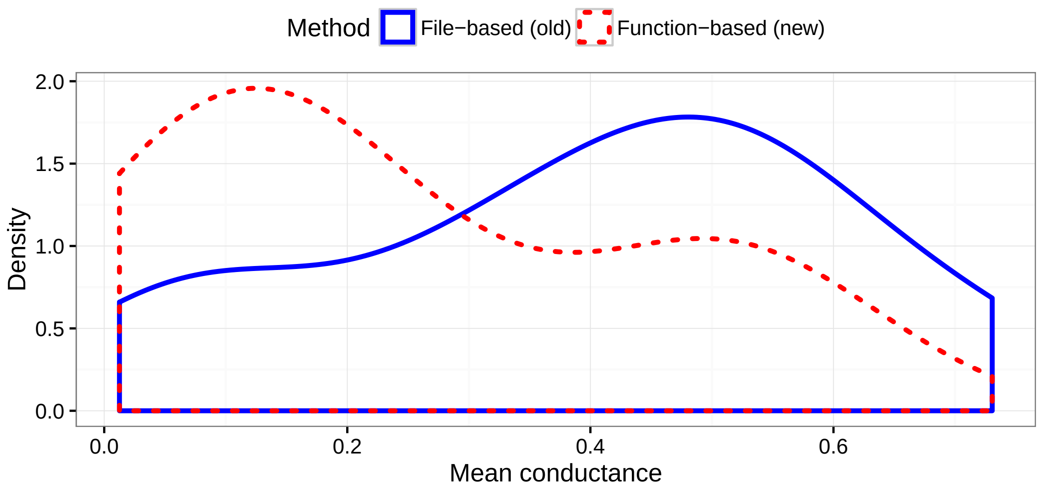


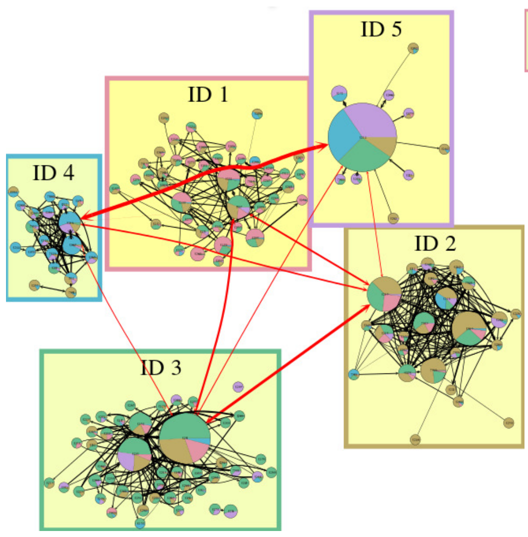
Figure 3.7: Density plot of mean community conductance computed for each project comparing the file-based and function-based methods.

strength of a relationship. We use PageRank centrality to identify important developers, denoted by the size of each node.[3]

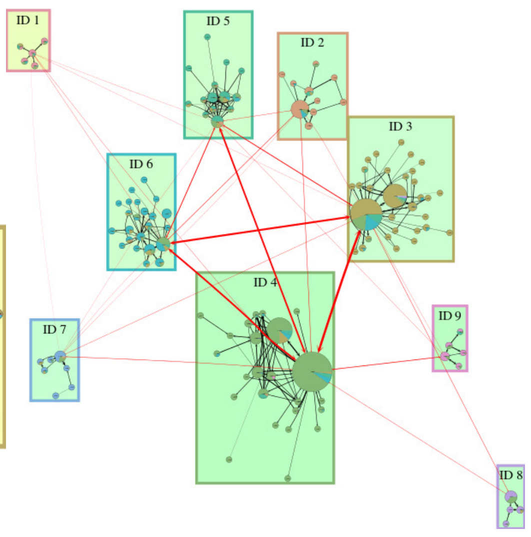
In Figure 3.8, *all* communities identified by the file-based method (left side) for QEMU are insignificant. The conductance of the communities is on the order of what is expected from a unorganized (rewired) network, represented by the yellow background color; it indicates that the file-based method failed to capture the organized structure of developer coordination. In contrast, the function-based method is capable of identifying several significant communities in the same project. Notice further that the file-based method has generated an extremely dense network, in which nearly every developer is contributing in every community, visible by the large number of multicolored nodes. In comparison, the developer network constructed using the function-based method is less dense; it has identified developers that make contributions only within one or two communities, which is visible by the large number of single-colored nodes.

In summary, we conclude that the file-based method fails to identify statistically meaningful communities as indicated by high conductance values that are statistically equivalent to the unorganized networks. In contrast, the function-based method was able to identify the latent community structure that

[3]To reduce clutter, we filter the inter-community edges by aggregating the edge multiplicity between two communities into a single edge, connecting the two most important developers. The weight of an inter-community edge represents the total coordination between all nodes in the connected communities.



(a) File-based developer network



(b) Function-based developer network

Figure 3.8: Developer networks constructed from QEMU v1.1. All communities in the file-based network are insignificant (yellow background color). The function-based method identified several significant communities (green background color).

was concealed by the file-based method. We emphasize that we use conductance here to evaluate whether the topological structure exhibits statistically significant communities, but we have not made any judgment about the communities quality beyond strictly topological features. Thus, we *accept H2*.

3.2.5 Tag-Based and Committer–Author-Based Network Similarity

To test hypothesis H3, we opt for an alternative approach to validating the network structure because we have access to a ground-truth network constructed using commit tags. We constructed developer networks for three revisions of the Linux kernel using the tag-based method (as provided by Linux's VCS) and the committer–author-based method (automatically constructed by us, as described in Section 3.1.1.2). We chose Linux as the *sole* test subject, because it enforces a strict tagging convention for every commit, which the other subject projects do not.

For each revision, we compute the similarity between the tag-based and committer–author-based networks using a graph matching strategy based on the Jaccard index [For10]. Figure 3.9 shows the results as a density plot. We see

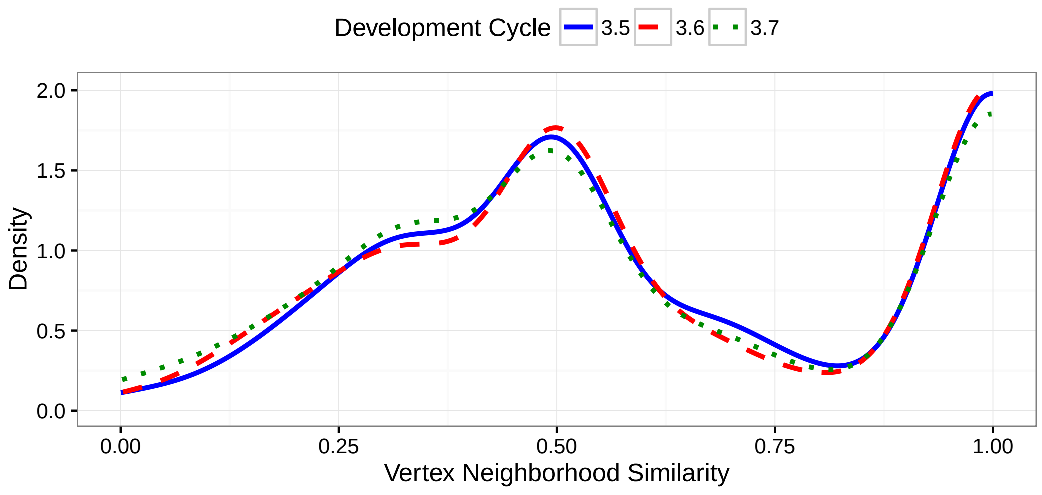


Figure 3.9: Comparison between tag-based and committer–author-based networks. Each curve represents a single revision of Linux. The similarity between each developer neighborhood is shown as a density plot with a mean of 70% similarity.

a bimodal distribution with peaks at 100% similarity (perfect match) and 50% similarity. The average taken over the three revisions is 70% match between the two networks. The probability of having a 70% match between two labeled random graphs, with equivalent size of the committer–author-based network, is less than a half of one percent; hence, we conclude that the committer–author-based network is an authentic representation of developer coordination, and we *accept H3*.

3.2.6 Network Validation

So far, we have not addressed whether the developer-network edges accurately represent real-world coordination, and if the detected communities have real-world significance. To address these unknowns, we conducted a developer survey to gather ground-truth data about developer coordination and communities.

Goals We now address whether the function-based method accurately captures real-world developer coordination by means of a survey. There is no need to include committer-author networks in the survey because they are constructed from direct references to developer coordination. Specifically, we address two research questions (RQ). First, we want to validate the network edges with respect to capturing real-world collaborative relationships according to developers who participate in the project.

RQ1—*Do the network edges, weights, and directions, accurately represent real-world collaborative relationships as they are understood by developers in the project?*

Second, we want to validate the real-world significance of the identified communities. Since community-detection algorithms are aware of only topological structure, there is no guarantee that the identified communities coincide with real-world communities that have vastly non-topological characteristics (e.g., common interests or development responsibilities).

RQ2—*Do the identified communities represent developer communities that have real-world significance?*

In Section 3.1.1.1, we identified that the function-based networks contain less edges than the file-based networks. What is not clear at this point is how those edges impact the developer communities. In particular, we are interested in the file-based edges that cross a community boundary (i.e. inter-community edges), since those are the ones that prevent the community detection algorithms from identifying the latent community structure.

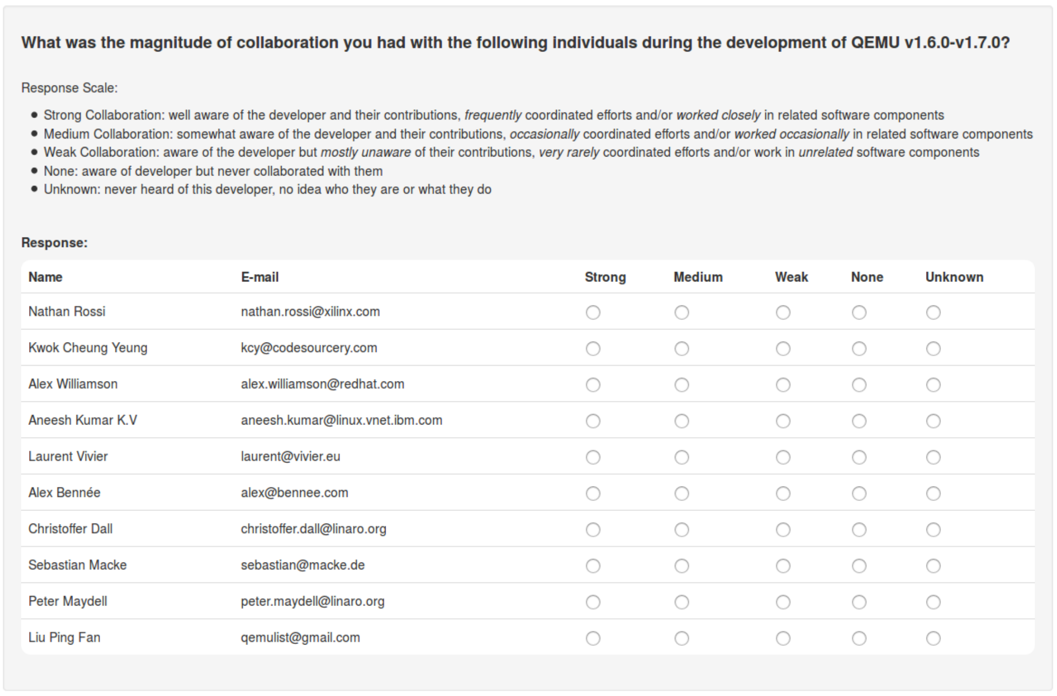
RQ3—*Are inter-community edges that are present in the file-based networks but missing in the function-based authentic with respect to developer perception?*

Participants We selected participants that made a contribution to one of our subject projects shown in Table 3.1 within the previous year, whose contact data we extracted from commit messages of publicly available VCSs. For each project, we constructed the developer networks and decomposed them into developer communities of, at least, five developers (fewer developers typically contain an insignificant number of coordinations). From the population of 6704 developers, we randomly selected 521 developers of different levels of involvement (e.g., single contributor to project lead), responsibilities (e.g., developers from various subsystems), and roles (e.g., tester, reviewer, bug fixer).

Experimental Material We conducted the survey online. It included, among others, demographic questions and the following survey questions (SQ):

SQ1.a—*Whom did you collaborate closely with during the development of version X?*

SQ1.b—*What was the magnitude of collaboration you had with the following individuals during the development of version X?*



What was the magnitude of collaboration you had with the following individuals during the development of QEMU v1.6.0-v1.7.0?

Response Scale:

- Strong Collaboration: well aware of the developer and their contributions, *frequently* coordinated efforts and/or *worked closely* in related software components
- Medium Collaboration: somewhat aware of the developer and their contributions, *occasionally* coordinated efforts and/or *worked occasionally* in related software components
- Weak Collaboration: aware of the developer but *mostly unaware* of their contributions, *very rarely* coordinated efforts and/or work in *unrelated* software components
- None: aware of developer but never collaborated with them
- Unknown: never heard of this developer, no idea who they are or what they do

Response:

Name	E-mail	Strong	Medium	Weak	None	Unknown
Nathan Rossi	nathan.rossi@xilinx.com	○	○	○	○	○
Kwok Cheung Yeung	kcy@codesourcery.com	○	○	○	○	○
Alex Williamson	alex.williamson@redhat.com	○	○	○	○	○
Aneesh Kumar K.V	aneesh.kumar@linux.vnet.ibm.com	○	○	○	○	○
Laurent Vivier	laurent@vivier.eu	○	○	○	○	○
Alex Bennée	alex@bennee.com	○	○	○	○	○
Christoffer Dall	christoffer.dall@linaro.org	○	○	○	○	○
Sebastian Macke	sebastian@macke.de	○	○	○	○	○
Peter Maydell	peter.maydell@linaro.org	○	○	○	○	○
Liu Ping Fan	qemulist@gmail.com	○	○	○	○	○

Figure 3.10: An example of a survey question to determine the real-world relationships between developers. The developers shown in this list are all sampled from the developer network so that we could compare each developer's response with the corresponding developer network.

SQ2—*Does the following network accurately represent collaborative relationships between developers?*

SQ3—*Do the developers shown in the above network represent a developer community?*

For SQ1.a, we provided auto-completion (obtained from analyzing the VCS) to help with recall and correct spelling of developers' names, for cross-referencing purposes. In a supporting question (SQ1.b), we presented a set of 10 developers sampled from the developer network and displayed the developers' name and e-mail address. An example of this is shown in Figure 3.10. For SQ2 and SQ3, we displayed a resizable visualization of the community network (not the entire developer network), labeled with the developer names. Both questions had to be answered on a five-point Likert scale [Lik32], ranging from strongly disagree to strongly agree. Additionally, participants could enter a free-format response where the participant could indicate any problems they experienced and elaborate on their responses. A pilot of the survey with ten testers did not reveal any significant issues.

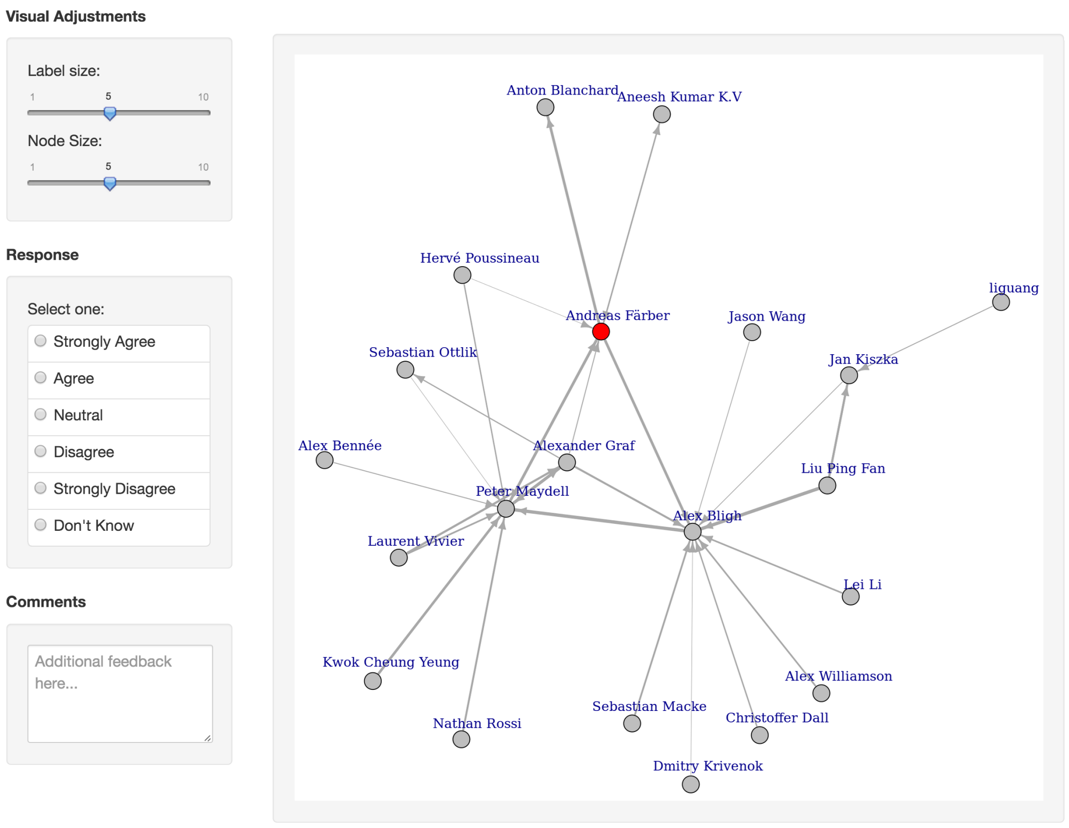


Figure 3.11: An example of the network visualization used in the developer questionnaire. The network represents a single developer community. The individual completing the questionnaire is represented by a solid red node.

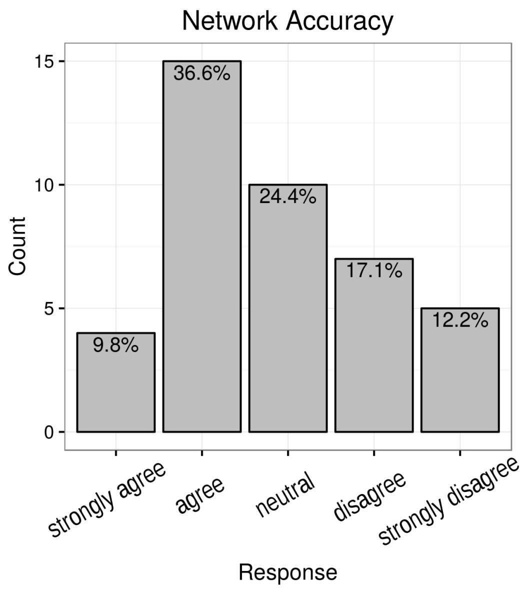
An example of the information that we provided to participants for the purpose of answering SQ2 and SQ3 is shown in Figure 3.11. The network shown in the questionnaire represents the developer community that the survey participant is a member of. The survey participant is represented by a solid red node, in this example the developer is Andreas Färber. Node labels indicate the identify of the developer based on information that we acquired from the VCS. The edge thickness in the visualization reflects the edge weight between two developers (i.e. strength of relationship). On the left side, the adjustments for the visualization are presented, the Likert scale response buttons, and the free-form text field.

Procedure To recruit survey participants, we sent a solicitation e-mail directly to the addresses found in the VCS. The solicitation e-mail contained an explanation of the goals of the survey in addition to a link where the survey could be found. The full text contained in the solicitation e-mail is provided in the appendix (page 189). Due to the nature of the survey delivery mechanism, the participants were able to fill out the survey in any environment they desired. We did not directly supervise the participants, and no time constraint was imposed. Upon completion of the survey, the participant electronically submitted the survey, and the responses were sent to our database where the participants information (e.g., unique id, e-mail address, cluster id etc.), project data (e.g., name and revision), and survey responses were automatically stored.

Analysis: Network Validity (RQ1) and Community Validity (RQ2) In total, 53 developers of the 521 that we contacted completed the survey. We show the responses to the Likert-scale questions in Figure 3.12. Regarding RQ1 (left in the figure): Almost half of the participants agree or strongly agree, and a quarter disagrees or strongly disagrees that the network accurately captured developer collaborative relationships. For RQ2, we see a similar distribution for the community-authenticity question (right in Figure 3.12): 53.9% of the participants strongly agree or agree that the communities have real-world significance.

Furthermore, we received a number of written responses for each question and categorized them manually. Regarding network accuracy, 13 written responses were given: 8 referenced missing developers or coordination, 3 referenced incorrect coordination, and 4 made various comments, such as, "Interesting Survey!". With respect to the validity of the edges, we received responses such as, "Many missing edges, but the ones that are present are ok." Regarding community authenticity, 14 responses were given: 8 stated that the network is accurate and provided a real-world meaning to the community, 2 responses stated that the network is accurate, 3 responses stated the identified communities are partially accurate, and 2 responses stated the network was inaccurate. Some example responses regarding community validity and interpretation are, "Most developers in the network are indeed people that I collaborate with." We also found evidence that the communities are representative of developers with related tasks. One developer wrote the following about a particular community: "All developers worked on the sfc driver, as employees or contractors of [company name]." Another wrote, "[developer name 1], [developer name 2] and myself work on similar system feature." Lastly, another participant wrote, "That [community] network is part of Kernel Media subsystem."

Analysis: Inter-community Edges (RQ3) We now address whether missing edges in the function-based network that are present in the file-based network (cf. Section 3.1.1.1) distort the view on coordination by neglecting authentic



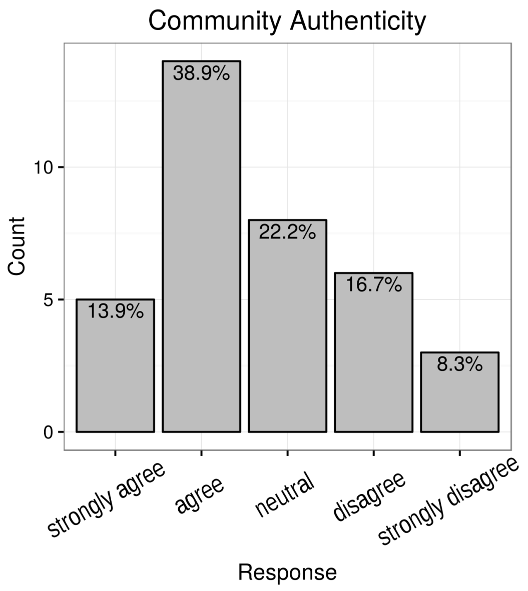


Figure 3.12: Developer survey responses to questions stated in Section 3.2.6 (SQ2 left, SQ3 right).

relationships. Unfortunately, we are unable to directly compare the file-based and function-based communities, because the file-based networks are extremely dense and hinder community-detection algorithms [JSS11]. Instead, we focus our attention to the edges that cross a community boundary, because these edges conceal the community structure and the results of Section 3.2.4 indicate that most missing edges are in fact cross-community edges. We used the responses from SQ1 to test the authenticity of cross-community edges neglected by the function-based method. To accomplish this test, we first identified the communities using the function-based method. We then used the function-based communities to identify all the edges that crossed community boundaries in the equivalent file-based network (i.e., same project and revision). We then removed all developers from the network who did not answer the survey. Finally, we calculated the percent of file-based cross-community edges that were confirmed by survey responses. For two subject systems (Linux and QEMU), we collected a sufficient amount of data (148 ground-truth edges). For QEMU, we acquired 47 ground-truth edges between 25 developers. Among the 25 developers, we found 82 file-based edges that cross a community boundary and were neglected by the function-based method. On average, 15.3% of the edges crossing a community boundary are authentic with a median of 7.7%. For Linux, we acquired 101 confirmed edges, and none of the 27 file-based edges crossing a community boundary were authentic. From these results we conclude that most of the edges that obscure the community structure in the file-based method are in fact unconfirmed by developers that answered the survey and appear to be an artifact of the method.

Interpretation & Discussion Both histograms in Figure 3.12 illustrate a substantial quantity of agreement responses for both the network-accuracy (RQ1) and community-authenticity (RQ2) questions. Additionally, in the written responses, we see that developers largely agree that the networks are accurate and that the identified communities have real-world meaning. We conceptualize the primary result using the information retrieval concept of precision and recall, using the terms defined in Figure 3.13, according to:

Edge Validity Terminology

	Confirmed	Unconfirmed
Present	Authentic Edge	Unauthentic Edge
Absent	Missing Edge	Not Applicable

Figure 3.13: An edge between a pair of developers is classified according to four possibilities. Confirmed means that a relationship is confirmed by a developer in the project. Unconfirmed means that no developer made any comment regarding the relationship. In the developer network, an edge may be present or absent. The Cartesian product of these sets produce the four categories. For example, a relationship that is confirmed and appears in the developer network is referred to as an authentic edge.

$$\text{Precision} = \frac{|\text{Authentic Edges}|}{|\text{Authentic Edges}| + |\text{Unauthentic Edges}|} \tag{3.11}$$

$$\text{Recall} = \frac{|\text{Authentic Edges}|}{|\text{Authentic Edges}| + |\text{Missing Edges}|}. \tag{3.12}$$

Interestingly, the responses indicate that our approach has good precision because the identified coordination relationships between developers and the developer communities are largely confirmed by developer perception, but the approach has imperfect recall, because some confirmed edges are absent from the developer network. Compared to the file-based method, we noted that the function-based method contains less edges (cf. Section 3.1.1.1). For Linux and QEMU, we were able to investigate the authenticity of missing edges that influence the community structure (RQ3). Overall, we found that very few file-based edges that are not captured in the function-based method were authentic. This means that the file-based method fails to exhibit significant community structures because it incorrectly connects members of different communities with edges that are not reflective of real-world relationships. In contrast, the finer-grained function-based method contains less of these erroneous inter-community edges of the file-based method so that we are able to detect significant community structure. In essence, the function-based method prioritizes precision over recall so that latent structure can be detected in the developer network. Given that our approach is fully automated and based on a single data source, this is a very encouraging result. To be fair, it was not to be expected that our approach achieves perfect recall rates, since not every mode of coordination is manifested in the VCS. Still, despite this lack of information, many developers agree that the identified communities capture a logical partitioning of developers.

3.3 Threats to Validity

External We applied our approach to ten manually selected subject projects, which threatens external validity. We chose projects of different but substantial sizes, with long and active development histories, and from different application domains to cover a considerable range of projects. Despite the diversity of our subject projects, they still represent only a fraction of the total diversity of software projects. The projects we considered appropriate for our study are mature and largely successful in many dimensions (e.g., highly active, with many developers and users). Meaningful community structure in developer networks may be limited to these kinds of projects. In unsuccessful projects, developers may be less aware of coordination requirements and so they are less reflective of real-world relationships between developers. Furthermore, it is unclear as to whether our results generalize to commercial projects since all of the subject projects are open source. Commercial project data is generally well protected, making such studies difficult. For the tag-based networks, only a few projects employ a strict tagging convention, so we cannot generalize the results.

Internal We realize that incomplete or incorrect recollection of a developer's collaborative relationships could compromise the survey responses. To help mitigate the consequences, we displayed the labeled developer network beside the survey questions. The compromise is that developers may only recall the collaborative relationships that are shown in the network and still forget others. However, this only influences the completeness and not the correctness of the responses. Since we already characterize our method as partially incomplete, the consequences of displaying the developer network does not affect our conclusions. A control group formed by including a secondary survey with randomized networks could increase confidence in the results, however, we recognized that the response rate is low and a control group would further reduce the already small experimental group size.

Construct The use of Ctags to identify indications of coordination requirements at the function level is based on heuristics. This leaves room for introducing misclassified lines of code by attributing them to the wrong function block. However, this threat to validity is minor, as only many misclassifications would influence the outcome of the statistical analyses we applied. As Ctags is widely used in practice, this is not to be expected. In a sense, we accept this minor threat in exchange for an approach that is language independent. To establish a weight on edges in the developer network, we use a metric based on the lines of code and the time in which the change was made. Lines of code is only one possible indicator of effort or involvement in the implementation of a function and it is unclear at this point whether other metrics (e.g., commit count) would be more appropriate. Nevertheless, the weights have played a central role in the identification of meaningful and valid community structure, which is a testament to the appropriateness of the edge weighting function for this particular purpose.

3.4 Related Work

Developer networks constructed from VCS data was first done by Lopez-Fernandez et al., where developers were linked based on contributions to a common module [LRGH09; LRG$^+$04]. Huang et al. improved Fernandez's work by automating module classification using knowledge of file directories [HL05]. The results indicated that modules may not provide detailed enough information to be useful. Improvements were made by narrowing the coordination assumption to common file contributions to identify more fine-grained coordination [JSS11]. Narrowing the definition of coordination helped to identify more subtle features than with the module-based assumption, but the authors noted that the networks were still too dense to identify community structure. In our approach, we use an even finer-grained definition of coordination to further reduce the density of the network, which enabled us to uncover community structures, in the first place. Previous work mostly applied metrics that do not produce rich visualizations, such as degree distributions or centrality plots, and no one has visualized community structure [HL05; LRGH09; LRG$^+$04; MW11; TMB10]. We are aware of one paper focusing on visualization of developer coordination using a file-based method, but community-detection was not possible without edge filtering due to the extreme density of the developer networks [JSS11].

Toral et al. applied social-network analysis to investigate participation inequality in the Linux mailing list that contributes to role separation between core and peripheral contributors [TMB10]. Bird et al. investigated developer organization and community structure in the mailing list of four open-source projects and used modularity as the community-significance measure to confirm the existence of statistically significant communities [BPD$^+$08]. Panichella et al. constructed developer networks based on mailing-list and issue-tracker data to identify developer teams and examine the driving forces behind splitting and merging teams during system evolution [PCDO14]. Our work differs by constructing networks from source-code contributions, instead of communication networks based on e-mail archives or issue trackers. Additionally, we apply our approach to a diverse set of projects and show that our findings have real-world significance.

Bird et al. examined the influence of code ownership on defect proneness at the component level of two commercial software products [BNM$^+$11]. They operationalize ownership based on the percentage of commits to a component made by a single developer, however, in open-source projects component ownership is rarely dominated by a single individual [WOB08]. Our work is complementary by supporting the identification of developer communities, which can be used to study ownership at the community level instead of the developer level.

Cataldo et al. examined the concept of socio-technical congruence and its impact on development productivity and software quality [CH13; CHC08]. Based on knowledge of work dependencies and technical dependencies, they identified coordination requirements. The “fit” between the actual coordination and required coordination was examined with the conjecture that high congruence is a desirable property. To establish the actual developer coordination, they used a-priori knowledge of developer teams, manual investigation of communication logs, and modification requests. Our work contributes to their framework by providing a fully automated method to identify modes of coordination using only data from the VCS.

Previous work utilized the Linux tagging convention to construct a developer network consisting of people involved in reviewing, acknowledging, and testing commits [BRB$^+$09]. We extended this work by proposing a method to extract similar information for projects that do not use the manual tagging convention, to automate the approach, and we validated it against the tag-based network for Linux.

Meneely et al. addressed the question of whether networks constructed from VCSs using the file-based method captured real-world coordination [MW11]. They concluded that the file-based networks were largely representative of developer perception, but that the networks suffered from errors in missing coordination and also falsely suggesting coordination. In contrast, our survey revealed that the more fine-grained method mainly suffers from missing edges. Furthermore, we extended on the original questionnaire format by allowing the participants to observe the developer network directly, instead of only displaying a list of names.

3.5 Summary

Task interdependencies, and consequently the need for developers to coordinate their modifications to source code, are intrinsic to software development. Information stored in the VCS provides valuable evidence of where task interdependencies exist by providing traceability between developers and source-code they modified. This information provides the basis for constructing developer networks that are a representation of the overall coordination structure for a software project. In order to derive value from these networks, they must be an authentic representation of the real-world. In this chapter, we proposed new approaches for constructing developer networks: (1) based on fine-grained information, by exploiting source-code structure to localize changes to source-code entities (cf. Section 3.1.1.1), and (2) based on commit meta data to identify committer–author relationships (cf. Section 3.1.1.2), which should, in theory, lead to more authentic developer networks compared to approaches that make use of only file-level information. We compared our fine-grained method to the existing file-based method (cf. Section 3.2.4) and compared our committer–author networks to tag-based networks that are constructed from manually reported “Sign-off” tags (cf. Section 3.2.5). We stated a set of hypotheses regarding the existence and statistical significance of community structure in developer networks (cf. Section 3.2.1). We then used developer communities as the basis for conducting a survey to determine whether our developer networks are valid with respect to developer perception (cf. Section 3.2.6). To address the hypotheses, we analyzed 10 open-source projects from a variety of domains, implemented in a number of different programming languages.

First, we found that the current file-based method for constructing developer networks from the VCS generates networks that are extremely dense. We found that by using a finer-grained heuristic, based on source-code structure, the network density is lower and the communities we are able to infer are stronger (according to conductance), compared to file-based developer networks (cf. Section 3.2.4). The statistical verification procedure (cf. Section 3.1.2.2) revealed that in the file-based networks, the communities are too weak to be statistically significant. In the case of the fine-grained developer networks, the communities are statistically significant, indicating that the developers are arranged according to the non-random organizational principle of modularity. In essence, the file-based heuristic obscures the self-organizing nature of software developers in open-source projects because, at this coarse granularity, developers appear to associate with each other randomly. In Section 2.3, we introduced the philosophy of a network perspective and that the primary interest is that of within-variable relationships that stem from tie dependences. In this regard, one major contribution made here is a new method for abstracting developer activities (the real-world phenomenon) into a network representation that exhibits tie dependence, a property that was missing from the file-based abstraction. Without tie dependence, there is little justification for applying a network abstraction to this particular phenomenon. With tie dependence comes the emergence of structural patterns that can be exploited for practical insights to support software development. For example, the pattern of communities helps us to understand where costly and risky inter-community coordination requirements exists that should be closely monitored and supported to avoid coordination failures.

Second, we found convincing evidence that the fine-grained developer networks align well with developer perception. The results of our survey of 53 developers indicate that our fine-grained approach is precise, in that the networks and communities are an authentic representation of the real-world, but is not a complete representation because some developer relations are missing (cf. Section 3.2.6). We were also able to show that the reason the file-based networks fail to exhibit latent community structure is because the method introduces many unauthentic edges that cross the community boundaries, which conceals the community structure (cf. Section 3.2.6). While no ground-truth developer network exists, by determining that fine-grained developer networks align well with developer perception, we have brought real-world meaning to the networks. In Section 2.3, we discussed how the process of abstracting real-world phenomena into a network representation often requires non-trivial leaps and that one must always remain cognizant and critical of assumptions that are made during the abstraction procedure to ensure real-world validity is not significantly compromised, distorted, or lost. Specifically in developer networks, there are a number of questionable assumptions that stem from the artifact granularity of existing approaches for constructing developer networks (cf. Section 2.4). In this chapter, we have demonstrated how to use source-code structure to reduce the strength of assumptions regarding relationships between developers' modifications to source code and how this improvement has a substantial impact on the appearance of higher-order structures in the resulting developer network and its real-world validity.

In this chapter, we have taken a strictly static view point on developer activities and relationships between developers. However, we recognize that many important socio-technical factors of software development are dynamical in nature (cf. Section 2.1). Now that we have a valid view on the coordination structure of software projects, we are able to add in the time component to study evolutionary principles of developer coordination. In the following chapter, we will continue to examine a number of non-random organizational principles (e.g., modularity, scale-freeness, and hierarchy) from an evolutionary perspective. This will shed light on how the developer organizational structure adapts as a project grows, how new developers enter a project, and how a developer's position in the network changes over time.

CHAPTER 4

Evolutionary Trends of Developer Coordination

This chapter shares material with the EMSE paper "Evolutionary Trends of Developer Coordination: A Network Approach" [JAM17].

In Chapter 3, we found that source-code changes provide valuable evidence of which developers are engaged in a coordinated effort. From this source of information, we constructed developer networks that represent the global developer coordination structure. In particular, we found that localizing source-code changes to source-code entities (e.g., functions), instead of files, has a substantial impact on the structure of the resulting developer network. Specifically, fine-grained developer networks, based on function-level information, exhibit a statistically significant community structure, and the network and corresponding communities largely coincide with developer perception. The weakness we identified was that some real-world relationships between developers are not expressed in the developer network, but that is balanced by the fact that edges that do appear in the network are authentic.

4.1 Motivation and Research Questions

The perspective we adopted in Chapter 3 was primarily a static one where we took snapshots of mature projects at a late stage of their development. A static view point is very limiting because change in software is inevitable, and the constant pressure to adapt to a changing environment is a major challenge that all software projects encounter. A primary source of pressure to change is a consequence of assumptions embedded in the software's design and implementation that become invalid over time and if a project fails to respond to adaptation pressure, the degree of satisfaction provided by the software decreases with time [LRW^{+}97]. While the pressure to change can arise from numerous sources, the influence of this pressure is not isolated to the software design and implementation, it permeates through all artifacts and facets of a project including the entire *organizational structure*. As the software evolves, the organizational structure building the software must also evolve to maintain effective coordination between developers. In the ideal case, a match or congruence is achieved between the coordination requirements implied by the project's technical artifact structure and the coordination mechanisms implied by the developer's organizational structure [CHC08]. In this chapter, we take on a dynamic view point by examining the evolution of developer coordination with respect to three fundamental organizational principles.

By gaining an understanding of the evolutionary patterns of developer coordination, software engineering practitioners will be in a better position to identify and respond to changing coordination requirements. For example, it is a well known phenomenon that adding developers to a project typically reduces overall productivity [Bro78; SMS15], but the precise mechanism behind this phenomenon is not yet well understood. It may be the case that adding developers to a project negatively interferes with coordination requirements by introducing additional complexity, which in turn causes decreases in productivity. To address this unknown, we need to better understand the effect of adding developers on the coordination structure. Once we have this understanding, we can establish processes for integrating new developers that minimize the decrease of overall productivity by minimizing the influence to the existing coordination structure.

4.1.1 Organizational Principles

In this chapter, we apply a sliding-window technique to transform discrete software changes, recorded in the version-control system, into a stream of evolving developer networks. Each developer network represents one snapshot in time, but the temporal distance between two snapshots is small as to capture the stepwise nature of evolution in the network topology. We construct the developer networks using fine-grained information to ensure that the network is an authentic representation of real-world developer relationships (cf. Chapter 3). We then examine the developer network with respect to the following three well-known and statistically well-founded organizational principles:

- **Scale freeness.** Scale-free networks are characterized by the existence of hub nodes with an extraordinarily large number of connections, which results in several beneficial characteristics including robustness and scalability (cf. Section 2.3.5). Developer networks of this kind are conjectured to tolerate substantial breakdowns in coordination without significant consequences to software quality [CH13; DM03].

- **Modularity.** The local arrangement of nodes into groups that are internally well connected gives rise to a modular structure (cf. Section 2.3.6). Modularity is a notable characteristic of many complex systems and indicates the specialization of functional modules [DM03]. In the case of developer organization, this is the primary organizational principle used to reduce system-wide coordination overhead and increase productivity [Bro78].

- **Hierarchy.** The global arrangement of nodes into a layered structure, where small cohesive groups are embedded within larger and less cohesive groups, forms a hierarchy (cf. Section 2.3.7). Hierarchy is an organizational principle distinct from modularity and scale freeness, and has been shown to improve the coordination of distributed teams [HM06]. For developer networks, hierarchy suggests the existence of stratification within the developer roles, and it indicates a centralized governance structure where decisions are primarily made at the top and passed down through a chain of command.

4.1.2 Research Questions

By means of a longitudinal empirical study on the evolution of 18 well-known open-source software projects, we will address the following two main research questions (RQ):

RQ1: Change—*What evolutionary adaptations are observable in the history of long-lived open-source software projects concerning the three organizational principles?*

RQ2: Growth—*What is the relationship between properties of the three organizational principles and project scale?*

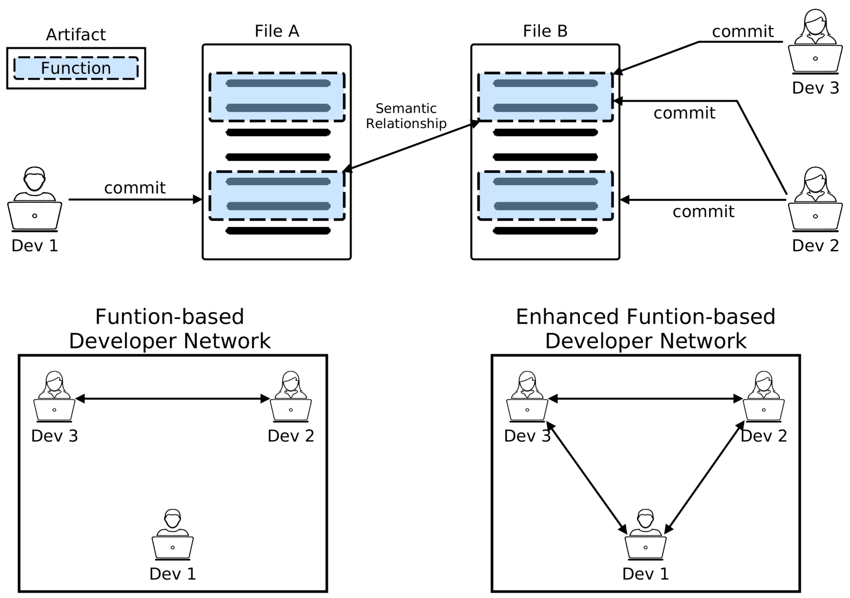


Figure 4.1: Three developers edit two semantically coupled functions in separate files (top). The resulting developer network from applying the original function-based construction method is shown bottom left. The resulting developer network from applying our enhanced construction approach that includes the coupling between artifacts is shown bottom right.

4.2 Methodology

4.2.1 Network Construction

In software projects, developers often work on distinct functions that are conceptually related to other functions that are written and maintained by other developers. For example, suppose that a developer provides a set of functions to perform logging of user information and a second developer makes extensive use of these functions for logging user triggered events in the graphical interface. To some extent the developers become constrained by each others development activities. If logging functions are changed substantially, the user of the functions may introduce bugs or unexpected behavior to the software. Therefore, a coordination requirement between the developers is implied by virtue of the relationship between the artifacts they change. A major disadvantage of the approach for constructing developer networks adopted in prior research [JSS11; JMA^{+}15; LRGH09; MRGO08; MW11; MWSO08] is that developer relations are solely based on mutual contributions to a single artifact so that higher-order developer coordination requirements, which stem from contributions to artifacts that are conceptually related but physically separated, will be omitted. This is particularly relevant for analyzing projects that are in an early phase of development because much of the implementation effort produces new code, which suggests there is a low likelihood of multiple developers contributing to the same function block. Since all phases of development are of importance for our study, we need a network construction approach that is appropriate for all phases of the project's development.

To capture a more complete model of the developer coordination, we augment the function-based developer network by enhancing it with semantic artifact-coupling information (cf. Figure 4.1). In Chapter 3, we demonstrated how the state of the art network construction approaches, which use a file-level heuristic, are too coarse gained to reveal the community structure of open-source developers. While our more fine-grained function-level heuristic is capable of capturing the community structure, and has shown evidence of reflecting developers' perceptions of the project's organization (cf. Section 3.2.6). However, the developer survey revealed that the main disadvantage of our approach is some missing edges in the network (cf. Section 3.2.6). By augmenting the developer network with semantic artifact-coupling information, we achieve a balance between the fine-grained approach, which misses edges when two developers work on separate functions, but we avoid the overly coarse-grain assumption of earlier file-level network construction approaches, which assume that all code within a single file is highly interdependent [JSS11; LRGH09; MRGO08; MW11; MWSO08].

Our approach is inspired by the socio-technical framework proposed by Cataldo et al. for identifying coordination requirements between developers working on interrelated tasks [CHC08]. While our implementation is one variation that uses a function-level artifact and semantic coupling, this approach naturally extends to accommodate other artifacts (e.g., configuration files, requirements, documentation etc.) and coupling mechanisms (e.g., dynamic, structural, co-change etc.). Figure 4.1 provides an illustration of the network-construction approach. Specifically, we reconcile the developer-artifact contribution (cf. Section 2.4) and software-artifact coupling information as follows: We begin by first identifying all developers' contributions to all functions in the system and express the contributions of M developers to N functions in an $M \times N$ matrix as

$$A_{\text{contrib}} = \begin{bmatrix} f(d_1, a_1) & \dots & f(d_1, a_N) \\ \vdots & \ddots & \vdots \\ f(d_M, a_1) & \dots & f(d_M, a_N) \end{bmatrix}, \tag{4.1}$$

where A_{contrib} is the function-contribution matrix and $f(d_i, a_j)$ represents contributions to artifact a_j by developer d_i. If a contribution to this artifact was made by this developer, then the element is 1, otherwise it is zero. Figure 4.2 depicts a situation where multiple developers make commits to multiple functions, some of which are coupled. With respect to Figure 4.2, all of the commit edges between developers and the functions they contributed to are expressed in A_{contrib}, and the coupling relationships between functions are expressed in another matrix described below.

We compute a matrix that represents the semantic coupling between artifacts using latent semantic indexing (LSI) (cf. Appendix A.3). We have chosen this specific approach because it has shown promising results in reflecting developers' perception of software coupling [BDO+13]. A detailed description of the approach is contained in the appendix (page 190). We represent the coupling for N artifacts in an $N \times N$ matrix as

$$A_{\text{coupling}} = \begin{bmatrix} \phi(a_1, a_1) & \dots & \phi(a_1, a_N) \\ \vdots & \ddots & \vdots \\ \phi(a_N, a_1) & \dots & \phi(a_N, a_N) \end{bmatrix}, \tag{4.2}$$

where A_{coupling} is the artifact-coupling matrix and $\phi(a_i, a_j) = 1$ if the two artifacts a_i and a_j are coupled and $\phi(a_i, a_j) = 0$ otherwise. In our study, this matrix represents the semantic coupling between functions. In reference to Figure 4.2, the coupling edges between all artifacts are expressed in A_{coupling}.

Finally, we combine the information contained in both matrices using the following operation

$$D_{\text{coord}} = A_{\text{contrib}} \times A_{\text{coupling}} \times A_{\text{contrib}}^{\top}, \tag{4.3}$$

where D_{coord} is the developer-coordination matrix, with elements that represent whether a coordination requirement between two developers exists. Intuitively, the matrix operation expressed in Equation 4.3 is computing the developers' mutual dependence based on contributions to common artifacts and artifacts that are semantically coupled. The resulting matrix D_{coord} is symmetric with respect to the principal diagonal.

We will now go through a minimal example to illustrate precisely how this procedure operates. Suppose that developer d_1 makes a commit to function f_1, developer d_2 makes a commit to function f_2 and f_3, and developer d_3 makes a commit to function f_3. Suppose also that the functions f_1 and f_2 are

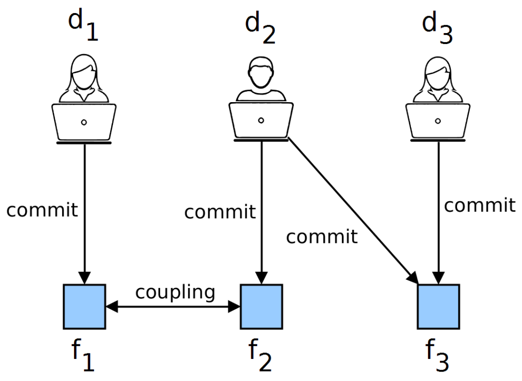


Figure 4.2: Three developers make commits to three different functions. Functions f_1 and f_2 are shown to have a coupling relationship between them.

semantically coupled. Figure 4.2 illustrates this particular situation. In our framework, the resulting contribution and coupling matrices are as follows.

$$A_{\text{contrib}} = \begin{array}{c|ccc} & f_1 & f_2 & f_3 \\ \hline d_1 & 1 & 0 & 0 \\ d_2 & 0 & 1 & 1 \\ d_3 & 0 & 0 & 1 \end{array} \qquad A_{\text{coupling}} = \begin{array}{c|ccc} & f_1 & f_2 & f_3 \\ \hline f_1 & 1 & 1 & 0 \\ f_2 & 1 & 1 & 0 \\ f_3 & 0 & 0 & 1 \end{array} \tag{4.4}$$

To compute the developer network, we combine the two matrices according to Equation 4.3.

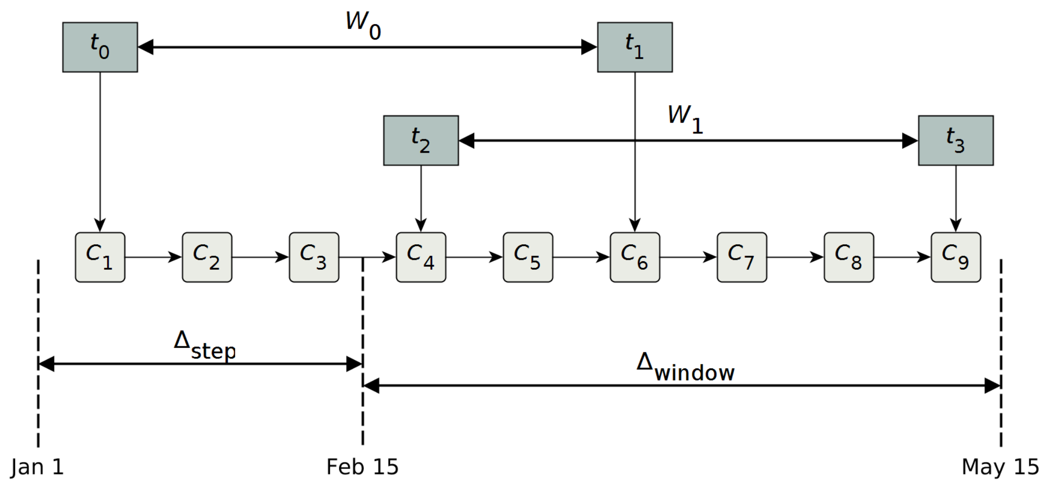
$$D_{\text{coord}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \tag{4.5}$$

$$= \begin{array}{c|ccc} & d_1 & d_2 & d_3 \\ \hline d_1 & 1 & 1 & 0 \\ d_2 & 1 & 2 & 1 \\ d_3 & 0 & 1 & 1 \end{array} \tag{4.6}$$

The adjacency matrix correctly expresses an edge between d_1 and d_2 because of their commits to coupled functions f_1 and f_2. Additionally, an edge exists between developers d_2 and d_3 because they both contributed to function f_3.

4.2.2 Developer Network Stream

In a second step, we capture the time-resolved evolution of developer organization by applying a graph-data-stream model to the network-construction



$W_0 = \{c_t \mid t \in [t_0, t_0 + \Delta_{\text{window}}]\} = \{c_1, c_2, c_3, c_4, c_5, c_6\}$

$W_1 = \{c_t \mid t \in [t_0 + \Delta_{\text{step}}, t_0 + \Delta_{\text{window}} + \Delta_{\text{step}}]\} = \{c_4, c_5, c_6, c_7, c_8, c_9\}$

Figure 4.3: A sequence of commits are shown in chronological order labeled $\{c_1, \ldots, c_9\}$. Two subsequent analysis windows denoted by W_0 and W_1 define which commits are included in each analysis window. The corresponding parameters for Δ_{step} and Δ_{window} used to define the sliding window process are also shown. Notice that both W_0 and W_1 include $\{c_4, c_5, c_6\}$ so that there is continuity in temporally close activities performed by developers over subsequent analysis windows.

procedure; a project's history is segmented into sequential overlapping observation windows, where each observation window captures a finite range of development activity. To linearize the development history, we flatten the master branch of the version-control system, which is essentially the linearization of a directed acyclic graph. All commits are then temporally ordered using the commit time. The n^{th} observation window is defined as a set W_n of commits, such that $W_n = \{\text{commit}_t \mid t \in [t_0 + n \cdot \Delta_{\text{step}}, t_0 + n \cdot \Delta_{\text{step}} + \Delta_{\text{window}}]\}$. Where commit_t is the commit occurring at time t, t_0 is the time of the initial commit, Δ_{window} is the window size, and Δ_{step} is the step size. A depiction of two subsequent analysis windows generated from the sliding-window approach is shown in Figure 4.3. Since software projects typically have long-term temporal trends (e.g., number of contributing developers), the evolution is temporally dependent and must be treated as a nonstationary process. This implies that the statistics (e.g., mean and variance of the metrics) will vary depending on when the project is observed. To properly analyze project evolution, we use a small enough observation window (90 days) for which the development activity has been shown to be quasi stationary [MW11]—a technique that is frequently employed in other domains with temporally-dependent processes [HSL+98]. To avoid artifacts that arise from aliasing and discontinuities between the edges of the observation windows, we opted for an overlapping-window technique [HSL+98] with a step size that is half of the window size. While smaller step sizes may be better, because of greater temporal resolution, we observed that using a smaller step size did not change the results, but did significantly add to the computational costs. In contrast, increasing the step size so that the windows did not overlap obscured periodic components in the data.

For each time window, we construct a network to represent the topology of developer coordination during a finite time range. The sequence of all finite windows generates the graph stream capturing the dynamic evolution of developer coordination over the entire project history. Each graph stream is then processed to extract a multivariate time series composed of the measurements that quantify the concepts of scale freeness, modularity, hierarchy, in addition to other context features, such as network size.

4.2.3 Scale Freeness

To identify a scale-free network, one must show that the degree distribution is described by $p(k) \propto k^{-\alpha}$, where $p(k)$ is the probability of observing a node with k connections and α as the power-law scaling parameter. In Section 2.3.5, we will discussed in detail how to determine the scaling parameter α and verify whether the model accurately describes the observed network.

When there are too few samples for statistical tests to be reliable, which is common early in the project history, we instead use the Gini coefficient [Atk70] to characterize the amount of inequality in the network's degree distribution. The Gini coefficient is bounded between 0 and 1, where 1 indicates strong inequality (i.e., possibly scale free); 0 indicates strong equality (i.e., not scale free). By definition, scale-free networks contain hub nodes, and as a result there is strong inequality in the distribution of edges connecting nodes. From this we conclude that a high Gini coefficient is a necessary condition for scale freeness, and if the network has a low Gini coefficient, it cannot be scale free.

4.2.4 Modularity

To quantify modularity, we use the well studied *clustering coefficient*:

$$c_i = \frac{2n_i}{k_i(k_i - 1)}, \tag{4.7}$$

where n_i is the number of edges between the k_i neighbors of node i [BLM+06]. The intuition is that $k_i(k_i - 1)/2$ edges can exist between k_i nodes, and the clustering coefficient is a ratio that reflects the fraction of existing edges between neighbors divided by the total number of possible edges (cf. Section 2.3.3). For example, if a node has a high clustering coefficient, then many edges exist between the neighbors of this node. Conversely, if a node has a low clustering coefficient, then only a few edges exist between neighbors of this node.

4.2.5 Hierarchy

Hierarchy manifests as a relationship between the node clustering coefficient and node degree (cf. Section 2.3.7). To test for the presence of hierarchy, we apply robust linear regression to solve for the optimal model parameters of the following model:

$$\log\left(\mathrm{CC}(v)\right) = \beta_0 + \beta_1 \cdot \log\left(\deg(v)\right), \tag{4.8}$$

where $\mathrm{CC}(v)$ is the clustering coefficient of vertex v and $\deg(v)$ is the degree of vertex v. If the optimal linear model has a nonzero slope (i.e., $\beta_1 < 0$) and the slope parameter is statistically difference from zero, such that $p < 0.05$ where p is that probability that $\beta_1 = 0$, one can conclude that hierarchy is present.

4.3 Study & Results

4.3.1 Hypotheses

We now present and discuss three hypotheses regarding the evolution of developer coordination networks. The hypotheses refine our research questions concerning the patters observed in the evolution of developer networks and the relationship between the patterns and project scale. The mapping between our research questions and hypotheses is the following: H1 and H3 are related to RQ1 and RQ2, H2 is related to RQ1.

H1—*Long-term sustained growth, in the number of developers contributing to the project, coincides with a scale-free developer coordination structure.*

It is almost folklore that adding developers to a project often has the opposite of the intended outcome, which is that adding developers will accelerate development [Bro78]. On this basis, there are important insights that can be gleaned from observing the changes that occur in the organizational structure as the project grows. Coordination of a large number of developers demands specialized coordination mechanisms, because the number of potential interactions among developers is quadratic in the number of developers [Bro78].

Additionally, the peripheral developer group, representing the vast majority of open-source software developers [NYN+02], are characterized by irregular, infrequent participation and high turnover rate [CH05; Koc04; MFH02; OJ07]. The implication is that a healthy developer network must be robust to node removals, which is the case for scale-free networks (cf. Section 2.3.5). Therefore, we expect that large developer groups self-organize into scale-free networks as an optimization for mitigating the coordination overhead and achieving resilience to coordination breakdowns (cf. Section 2.3.5). Following the reasoning of Brooks [Bro78], as the developer network grows, we expect, at some point, the developer count should stagnate or decrease, because of ineffective coordination leading to a loss of productivity and developers' motivation to participate. It has been shown that, in some situations, Brooks' law does not apply to open-source software projects [Koc04], and we hypothesize that scale freeness is a reasonable principle to explain this observation. Therefore, we expect very large projects to exhibit the scale-freeness property as a mechanism to maintaining productivity despite the potentially enormous coordination costs and risks imposed by a large but unstable peripheral developer group. Finally, scale freeness is an emergent property of a self-organizing system that is motivated by necessity. Since small developer groups do not benefit from a scale-free network structure as much as large developer groups, we do not expect small projects with a small number of developers to form scale-free networks. If scale freeness is required for sustainable project growth, this information helps us to identify healthy growth profiles in large projects. Practitioners can make use of this information to establish policies that encourage the addition of developers to a project in a similar manner as preferential attachment (cf. Section 2.3.4) so that the organizational structure reaches a scale-free state.

H2—*Developers initially form loosely connected groups that are not internally well connected (i.e., that have a low modularity). As time proceeds, developer groups tend to become more strongly connected in terms of the clustering coefficient until an upper bound is reached.*

As a project evolves, several factors encourage developers to coordinate, but there are also opposing forces. Based on prior experience and empirical evidence, software evolution tends to cause an increase along several project dimensions (e.g., lines of code, complexity, number of developers etc.) and will demand increasing levels of coordination between developers to avoid system degradation [LRW+97; LR01]. Furthermore, it is reasonable to expect that developers will become more familiar with each other and rely on the external knowledge of others for support in the completion of development tasks. Empirical evidence from studies on various open-source software projects also suggests that developers tend to specialize on particular artifacts (e.g., subsystems or files) and form groups with common responsibilities and shared mental models [JMA$^+$15; Koc04]. These influences increase modularity in the developer network by causing additional edges to form in local sub-networks that are dedicated to a particular responsibility. The opposing force arises from the quadratic scaling between the number of developers and potential coordination relationships, where the cost of coordination can easily dominate the benefit achieved from coordination [Bro78]. Therefore, developer coordination is constrained to evolve in a manner that balances these opposing forces. We expect that an equilibrium exists between the benefit and cost of coordination, and this will govern the evolution of developer coordination. If this hypothesis is confirmed, it suggests that the socio-technical environment in which developers work changes substantially over time. As a result, there are presumably changes in the coordination challenges that developers face and they would likely benefit from different techniques and tools to support coordination during different phases of the project.

H3—*In early project phases, the developer-coordination structure is hierarchically arranged. As a project grows and matures, the developer-coordination structure will gradually converge to a network that does not exhibit hierarchy, as the command-and-control structure becomes more distributed.*

A project's command-and-control structure is responsible for directing the work of others in a coordinated manner. In the early phases of a project, it is conceivable that the small number of initial developers have a comprehensive understanding of the global project details and are capable of effectively coordinating the work with others in a centralized configuration. In these early project conditions, hierarchy is an effective organizational structure because it promotes efficiency through regularity and is appropriate when the developer network is stable [Kot14]. As the project evolves and grows in the number of developers and system size, developer coordination becomes increasingly formidable, especially, once the peripheral developer group has grown to be significantly larger than the core developer group [CH05; Koc04; MFH02]. Empirical evidence indicates that efficiency in large open-source software projects is the result of self-organizing cooperative and highly decentralized work [Koc04], which becomes increasingly important as a project grows. The result is that the command-and-control structure must evolve to become more distributed, because no single person could reasonably have a comprehensive understanding of the global project state, and distributed self-organization must take over. Furthermore, hierarchy is an intrinsically inflexible organizational structure that strongly promotes regularity [Kot14], but as the project evolves, organizational flexibility becomes increasingly important so that the project can avoid the detrimental misalignment of organizational structure and the technical structure as a result of evolution [SER04]. For practitioners, it is often unclear which organizational structures are suitable for different project conditions. It may be the case that different organizational structures are more appropriate during early project conditions and others during late projects conditions. By addressing this hypothesis we gather evidence of how successful open-source software projects evolve with respect to hierarchy.

4.3.2 Subject Projects

For the purpose of our study, we selected 18 open-source software projects as listed in Table 4.1. The subject projects vary in the following dimensions: (a) size (source lines of code, from 50 KLOC to over 16 MLOC, number of developers from 25 to 1000), (b) age (days since first commit), (c) technology (programming language, libraries), (d) application domain (operating system, development, productivity, etc.), and (e) version-control system used (Git, Subversion). We chose these projects because they are all widely deployed, and have long development histories.

Table 4.1: Overview of subject projects

						Developer count		
Project	Domain	Lang	Period	SLOC	Commits	Cur.	Max	Min
Apache HTTP	Server	C	05/99–06/15	2M	73K	13	26	2
Chromium	User	C/++, JS	07/08–06/15	16M	533K	642	1056	71
Django	Devel	Python	07/05–01/15	400K	38K	98	105	3
Firefox	User	C/++, JS	03/98–06/15	12M	230K	417	474	62
GCC	Devel	C/++	06/91–01/15	7M	137K	117	122	2
Homebrew	User	Ruby	05/09–06/15	100K	42K	473	525	3
Joomla	CMS	PHP	09/05–06/15	400K	20K	53	78	2
jQuery	Devel	JS	03/06–06/15	65K	12K	5	30	2
Linux	OS	C	04/05–05/15	17M	570K	1445	1512	481
LLVM	Devel	C/++	06/01–06/15	1.2M	120K	127	128	3
Mongo	Database	C/++, JS	10/07–06/15	600K	28K	45	53	2
Node.js	Devel	C/++, JS	04/09–05/15	5M	23K	19	53	2
PHP	Devel	PHP, C	04/99–05/15	2.5M	100K	46	66	9
QEMU	OS	C	11/05–06/15	1M	37K	116	157	2
Qt 4	Devel	C++	03/09–04/15	1.5M	36K	7	122	5
Rails	Devel	Ruby	11/04–06/15	200K	49K	146	213	2
Salt	Devel	Python	02/11–06/15	200K	44K	204	205	3
U-Boot	Devel	C	12/02–06/15	1.2M	32K	114	134	2

4.3.3 Scale Freeness

We now discuss the results of applying the procedure described in Section 4.2.3 to address H1. The primary goal is to determine whether a power-law degree distribution is a plausible model for describing the observed developer networks and thus can be characterized as scale-free networks. We must eliminate Apache HTTP from this evaluation because the project has too few developers, and the statistical error with small sample sizes can lead to inaccurate conclusions. For the remaining 17 projects, if the goodness-of-fit p value is greater than 0.05, we can confidently conclude that the network is scale free.

Temporal Dependence One primary finding, which is true for all 17 subject projects, is that the scale-freeness property is temporally dependent, which means that this property is not universally present with respect to time. This is notable because it is a distinctly different view point from prior studies that approached the topic of scale freeness from a temporally static perspective [LRGH09]. To illustrate this result, a typical chronological profile is shown in the left portion of Figure 4.4, taken from LLVM. The top figure illustrates the network growth in terms of the number of developers contributing to the project, and the bottom figure illustrates the Gini coefficient. Each sample point represents a measurement for a single developer network that is computed for a single development time window. The shape of the sample point represents whether the network is scale free during the given development window. To help draw attention to the general trends in the data, a smooth curve has been fitted using locally weighted scatterplot smoothing, with 99% confidence intervals in gray. The evolutionary profile of a project is typically composed of the following three distinct temporal phases.

Phase I: Stagnation The initial phase, which can last for a number of years, is characterized by extremely limited growth in the number of contributing developers. In Figure 4.4, for LLVM, this phase occurs from the project's beginning in 2002 until 2006. During this period, the network exhibits high levels of coordination equality, because most developers are similar with respect to the degree of coordination with other developers and so the coordination requirements are uniformly distributed among all developers. The magnitude of coordination equality in the network is quantified using the Gini coefficient of the corresponding degree distribution. This is shown in Figure 4.4, for LLVM, where we see an initially low Gini coefficient (i.e., high equality). A low Gini coefficient indicates that, during this initial phase, most developers are similar in their degree of coordination with others, and the network is not scale free (i.e., it lacks the characteristic hub nodes discussed in Section 2.3.5).

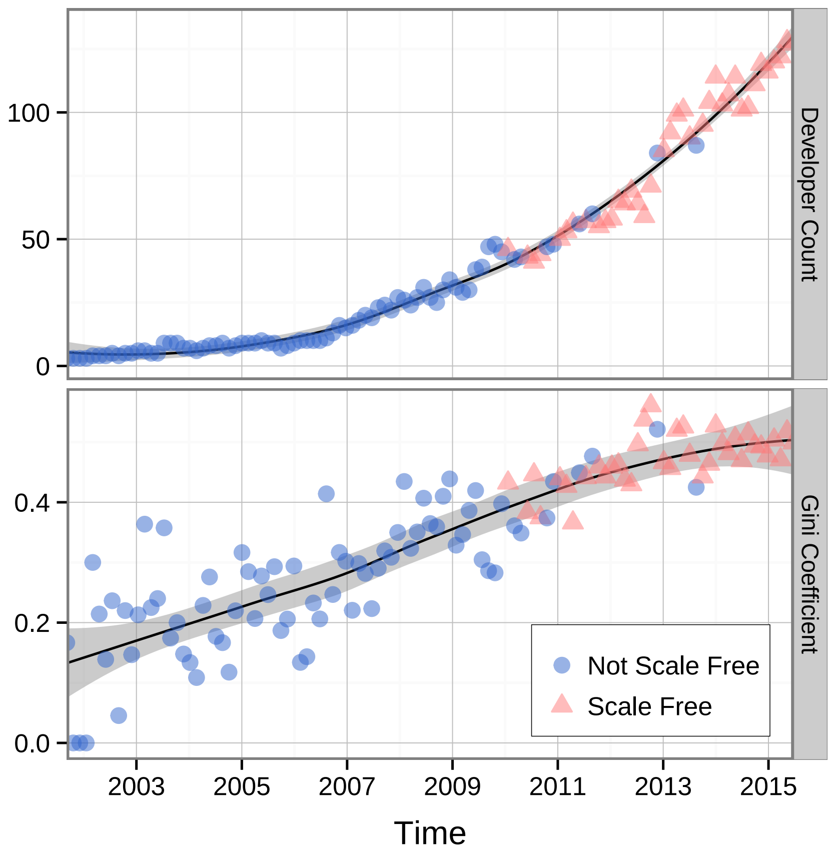


Figure 4.4: Evolutionary profile for entire history of LLVM. Time series are shown for the Gini coefficient (bottom) and the number of developers (top). A smooth curve is fitted to the observations with the 99% confidence interval shown in gray. The shape of the data points indicates whether the network was scale free for a given point in time.

Phase II: Transition In the second phase, we see that projects reach a critical mass point, at which a positive trend component is visible in the number of contributing developers. For LLVM (cf. Figure 4.4), this transition point occurred in late 2006 to early 2007 and the second phase lasts until mid 2011. Following the transition point, super-linear growth with an increasing slope occurs until the end of the analysis period. During that time, the Gini coefficient also has a positive trend component, indicating that the network has progressively less equality, because hub nodes, with significantly more coordination requirements than the average developer, begin to form. During this phase, the scale-freeness property emerges for the first time, but the state of being scale free is initially unstable. Most of the projects become scale free

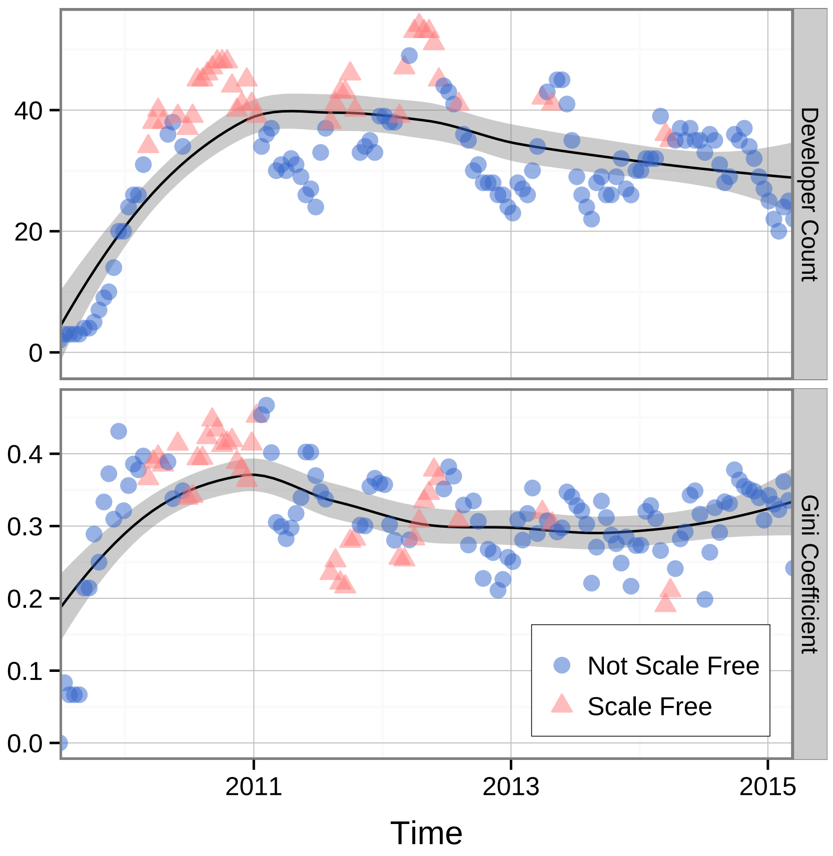


Figure 4.5: Evolutionary profile for entire history of Node.js. Time series are shown for the Gini coefficient (bottom) and the number of developers (top). A smooth curve is fitted to the observations with the 99% confidence interval shown in gray. The shape of the data points indicates whether the network was scale free for a given point in time.

once the network size has reached roughly 50 developers, but in *none* of the subject projects does a network exceed a size of 86 developers without first becoming scale free. The number of developers contributing to the project when the first appearances of scale freeness occurs is shown in Table 4.2 for all projects under the column label "SF Size". Overall, the two important results from this phase are that, during time periods with much less than 50 developers, developer equality is relatively high and scale freeness is not a common property. In contrast, during periods that significantly exceed 50 developers, developer networks are predominantly scale free. Essentially, the scale-freeness property appears to be dependent on the network size and the time of observation.

Phase III: Stabilization In 12 projects (Chromium, Django, Firefox, GCC, Homebrew, Joomla, Linux, LLVM, QEMU, Rails, Salt, and U-Boot), a third phase is visible in which the scale-freeness property stabilizes and is rarely lost. In Figure 4.4 for LLVM, this phase begins shortly after mid 2011 and extends until the end of the analysis period. By means of a visual inspection of the time-series data, a number of patterns is clearly visible. In all of the projects that achieve stable scale freeness (i.e., a scale-free state that is maintained over several consecutive analysis windows), they demonstrate the capability of long-term sustained (e.g., over the period of several years), and often accelerating, growth in the number of contributing developers. This is visible in Figure 4.4 for LLVM, where we observe an increasing slope in the number of developers year over year, and growth continues on until the end of the analysis period. In the other 6 projects (Apache HTTP, jQuery, Mongo, Node.js, Qt 4, and PHP), scale freeness is either never achieved, remains unstable indefinitely, or is lost indefinitely. The growth profile in these 6 projects is very different from the projects that achieve stable scale freeness and growth appears to be unsustainable because project growth rates decrease with time and often the number of contributing developers even drops. An example of these two distinct cases is presented in Figure 4.4, where LLVM reaches stable scale freeness and has long term accelerating growth. In contrast, Node.js, presented in Figure 4.5 does not achieve stable scale freeness and has unsustainable growth with long-term loss of developers. The percentage of time each project spends in a scale-free state is shown in Table 4.2 under column "% SF". The measurements indicate that the large projects that have had long term growth spend a significantly larger percentage of time in a scale-free state in comparison to projects without long-term sustained growth.

Degree Cut-off A scale-free network is characterized based on whether the *tail* of the degree distribution is described by a power law (cf. Section 4.2.3). A rarely addressed yet important factor, though, is the proportion of nodes that are described by the power law. We illustrate this point in Figure 4.6, where the tail region described by the power law excludes the majority of developers of the Linux kernel. We found that, in all projects, the proportion of developers that are described by the power law is low and typically ranges between 20%–50%. We saw that, there is an inverse relationship between the network size and the percentage of developers that are characterized by a power law. We illustrate the results for all projects in Table 4.2, where column "k_{min}" represents the lower bound for the power-law distribution, column "% Dev SF" represents the percentage of developers the are described by the power-law distribution, and column "SF Dev" represents the absolute number of developers described

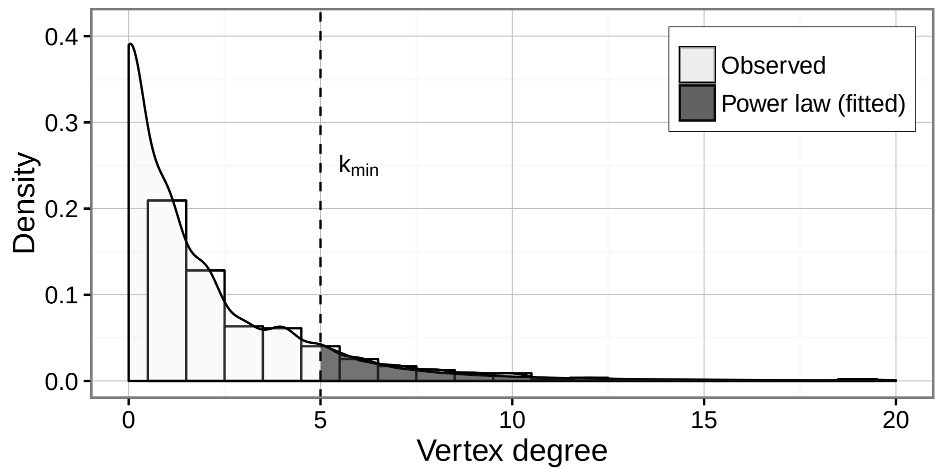


Figure 4.6: Power law fitted to the degree distribution for Linux. The power-law distribution describes the developers with a degree greater than k_{min}, the majority of low degree developers are not described by a power law.

by the power-law distribution. These measurements are taken from the most recent analysis period and the corresponding analysis windows are provided in Table 4.2.

Outliers We found two interesting outlier projects with respect to the presence of the scale-freeness property. For PHP and jQuery, the network size reached a peak at roughly 50 developers for several months yet never reached a stable scale-free state. After the peak, both projects then experienced years of continuous loss of developers and never recovered (cf. Figure 4.7). Another

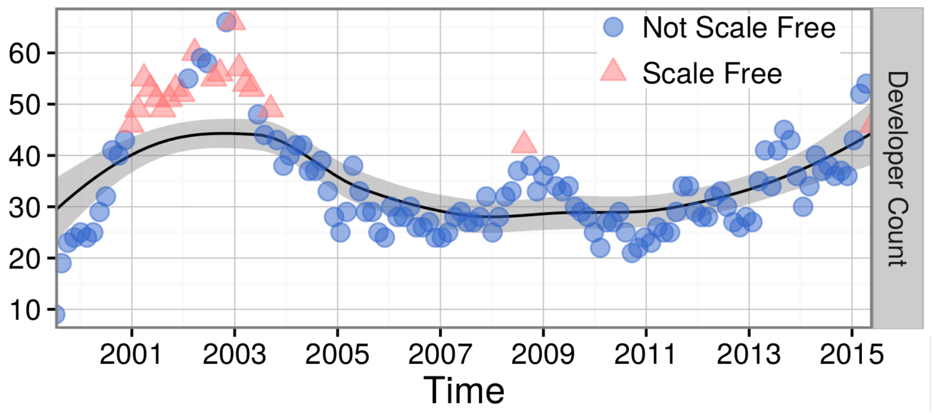


Figure 4.7: Evolutionary profile for entire history of PHP.

interesting outlier is Firefox, where during the period of September 2009 to December 2009 the network was frequently and unexpectedly not scale free. While the definitive cause of the disruption is unknown, we learned that, during this period, Firefox experienced severe release problems resulting in a major revision being released one year late.[1] It is interesting to note that the network-structure disturbances were observable several months before the first public announcement of release problems.

In summary, our study suggests that open-source software projects lack a scale-free structure in the initial phases while the developer network scale is small. In developer networks where growth significantly exceeds 50 developers, we always observe the emergence of scale freeness and no project ever grew beyond 86 developers without first becoming scale free. The caveat is that the scale-freeness property is temporally dependent, and in some projects, remains in an oscillatory state indefinitely. Overall, we *accept H1*.

[1] `http://www.cnet.com/news/mozilla-pushes-back-firefox-3-6-4-0-deadlines/`

Table 4.2: Developer network structural measurements

	Scale Free					Modularity		Hierarchy	
Project	S.F. size	%S.F.	k_{min}	% Dev S.F.	# Dev S.F.	μ_{cc}	σ^2_{cc}	$\beta_{1_{early}}$	$\beta_{1_{late}}$
Apache HTTP	N/A	0.00	7	0.46	6	0.49	0.12	-1.14	-0.85
Chromium	71	97.60	1012	0.08	53	0.44	0.04	-0.40	-0.35
Django	42	25.00	49	0.60	59	0.57	0.04	-2.32	-0.51
Firefox	73	90.60	481	0.07	30	0.48	0.05	-0.43	-0.27
GCC	36	55.90	164	0.26	30	0.43	0.04	-0.73	-0.40
Homebrew	80	87.50	355	0.49	230	0.61	0.02	-1.28	-0.29
Joomla	55	20.00	2	0.66	35	0.38	0.16	-1.67	-0.66
jQuery	30	1.41	5	0.80	4	0.47	0.10	-2.09	-1.66
Linux	515	98.80	1521	0.05	69	0.58	0.05	-0.28	-0.25
LLVM	46	32.70	51	0.24	30	0.45	0.08	-2.22	-0.39
Mongo	51	3.28	30	0.67	30	0.38	0.07	-2.30	-0.53
Node.js	35	16.30	1	0.47	9	0.16	0.10	-2.30	-1.33
PHP	46	14.60	29	0.65	30	0.45	0.08	-0.85	-0.44
QEMU	37	61.00	88	0.27	31	0.53	0.05	-2.19	-0.35
Qt 4	86	43.80	3	0.86	6	0.68	0.12	-0.50	-0.92
Rails	38	69.20	55	0.21	30	0.58	0.08	-2.17	-0.34
Salt	32	82.20	89	0.36	74	0.55	0.07	-0.95	-0.31
U-Boot	41	64.60	41	0.58	66	0.60	0.05	-1.39	-0.29

S.F. size – network size at first appearance of scale freeness
% S.F. – percent of time the project exhibits scale freeness
$k_{\min}$ – minimum bound on the power-law distribution
% Dev S.F. – percent of developers described by the power-law distribution
# Dev S.F. – number of developers described by the power-law distribution
μ_{cc} – mean value of clustering coefficient for latest development cycle
σ^2_{cc} – variance of clustering coefficient for latest development cycle
β_{1early} – slope parameter for an early development cycle
β_{1late} – slope parameter for the most recent development cycle

4.3.4 Modularity

The clustering coefficient is a means to measure the extent to which developers form cohesive groups. In Figure 4.8, we present the evolution of developer networks with respect to their clustering coefficients for all subject projects. The evolution of each project is illustrated by a time series that represents the mean clustering coefficient with a light gray boundary to indicate the 99.5% confidence interval. There is one evolutionary profile, in particular, that describes the majority of the subject projects. This profile is characterized by a positive trend component (i.e., increasing clustering coefficient) that smoothly converges to a clustering coefficient range of 0.45–0.55. For the projects that do not fit this profile, the positive trend component is not observable, possibly because we do not have a complete project history. For example, the development of the Linux kernel was started in 1996, but the publicly available Git repository only has commits dating back until early 2005. In Table 4.2, we present the results of the mean clustering coefficients (column "μ_{cc}") and variances (column "σ^2_{cc}") for the most recent revisions of each project.

The only project which does not closely adhere to the general pattern of convergence is Qt. The exceptional behavior seen in Qt is likely a consequence of the significant decrease in the number of active developers and possibly represents an evolutionary anti-pattern. The number of developers contributing to Qt is high until 2011 (cf. Figure 4.9), but this period is followed by several years of rapid decline. Similarly, we see that the mean clustering coefficient profile fits the general pattern until 2011, where the value suddenly drops and then oscillates before a radical upswing. It is worth noting that Qt is the only subject project exhibiting this pattern, and it is similarly the only subject project that has had such significant decline in number of contributing developers.

For the majority of the projects, the fact that they do not ever significantly exceed a clustering coefficient of 0.55 suggests that there is a limitation to the distribution of coordination requirements in the local developer neighborhood. To give a reference point, a clustering coefficient of 0.5 for a node means that there are edges between half of the nodes neighbors. We observe that there is a preference to achieve a state where roughly half of every developer's neighbors also have a coordination requirement (i.e., an edge). The evolutionary profiles of our subject projects indicate that developer networks evolve according to a process that promotes coordination requirements increasing up to a maximum, but that prevents the formation of coordination requirements between too many developers. There appears to be no non-zero lower bound on the clustering coefficient, but in all cases, an initially low clustering coefficient does tend to converge towards a clustering coefficient of 0.5. The implication of a bound is

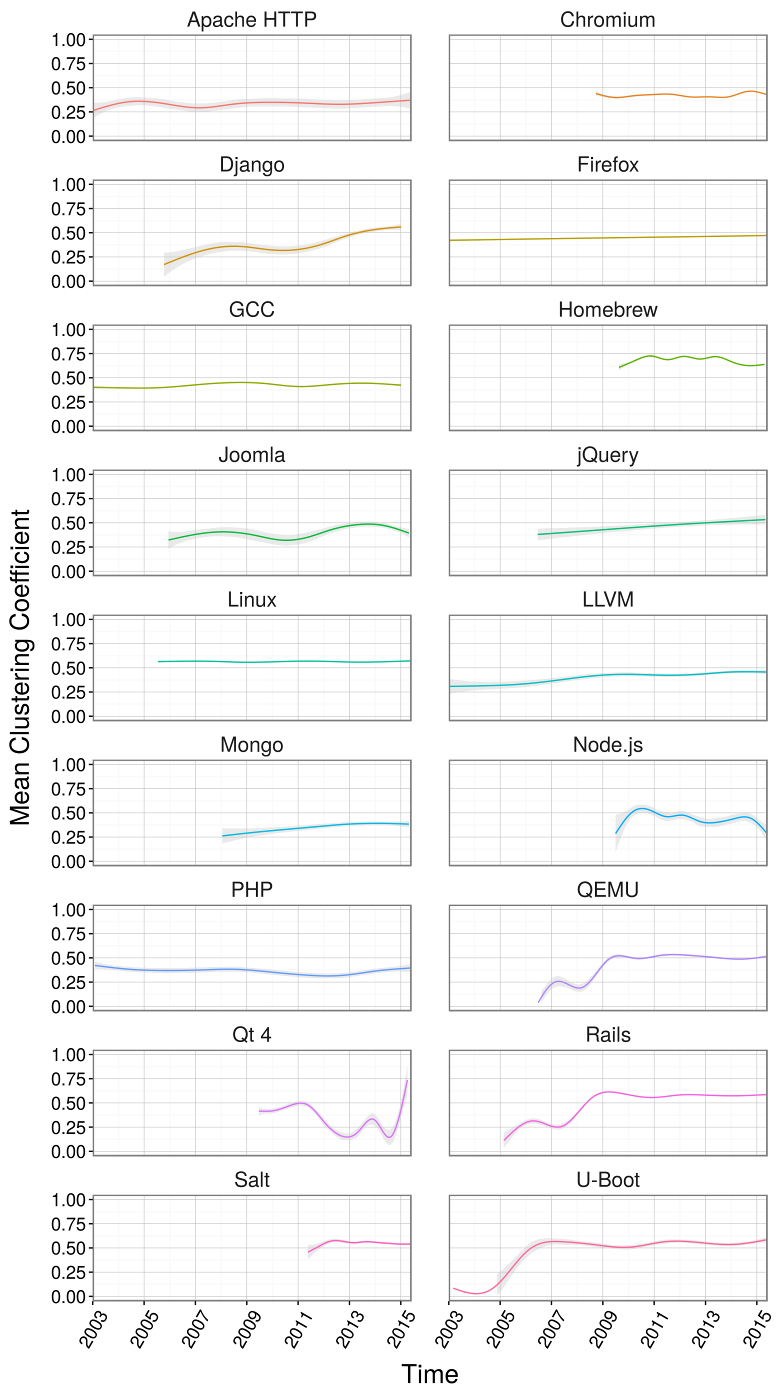


Figure 4.8: Clustering-coefficient time series for all subject projects. The light gray area indicates the 99.5% confidence interval.

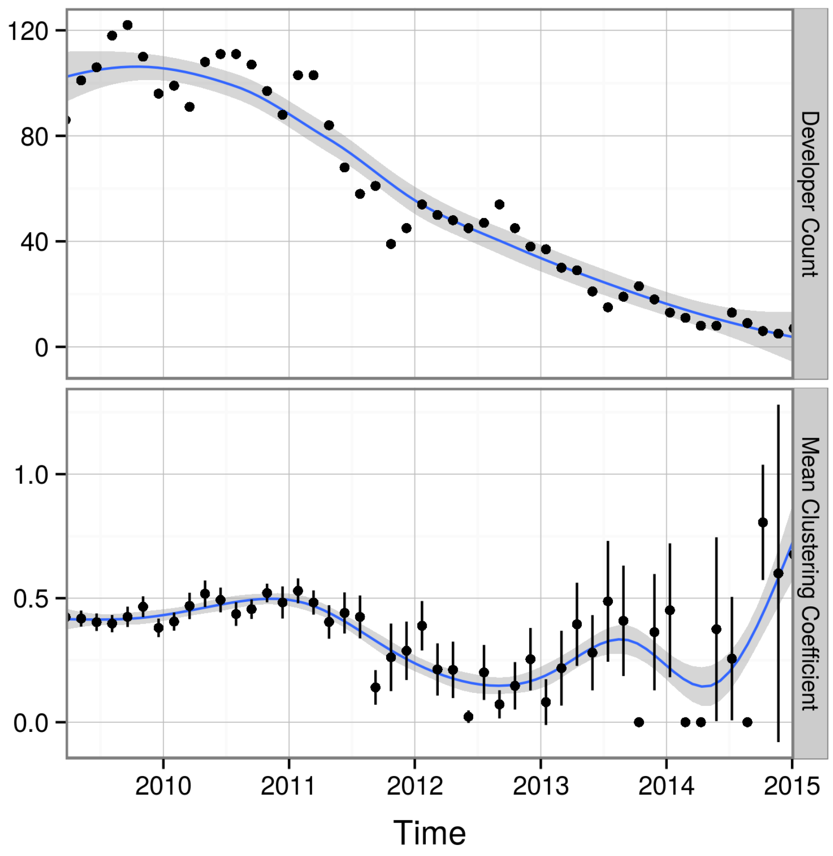


Figure 4.9: Developer count and mean clustering coefficient evolution for Qt with the standard error bars for the mean clustering coefficient included. The significant decline in developer count coincides with instability in the clustering coefficient, indicating that departing developers have a significant impact on the local connectivity of the developer network.

that no observation, under any circumstance, should ever violate the bound by crossing it. In this sense, we are not able to prove that the upper limit that we observed will never be violated. The statement that we are able to make at this point is that the observations made on these 18 subject projects are evidence that a bound likely exists. Particularly, the smooth convergences (i.e., a gradual decrease in the first derivative) to this upper limit is indicative of a bound this is not likely to be crossed in the future.

To better understand the mechanism behind the tendency for developers to form cohesive groups, we examined the relationship between network size (i.e., number of developers) and clustering coefficient. In all cases, we found that the clustering coefficient increases with the network size, however, this dependency decreases as the network size increases. This relationship is shown for subject project Django in Figure 4.10, which illustrates a roughly logarithmic relationship between network size and clustering coefficient. This is a notable result because, in the ER random graph model described in Section 2.3.4, the clustering coefficient decreases with network size, and in many real-world networks, clustering coefficient and network size are independent [AB02]. From this observation, we conclude that, in terms of modularity, developer networks form groups according to a non-random organizational principle that is also different from the preferential-attachment model used to explain many real-world scale-free networks (cf. Section 2.3.4).

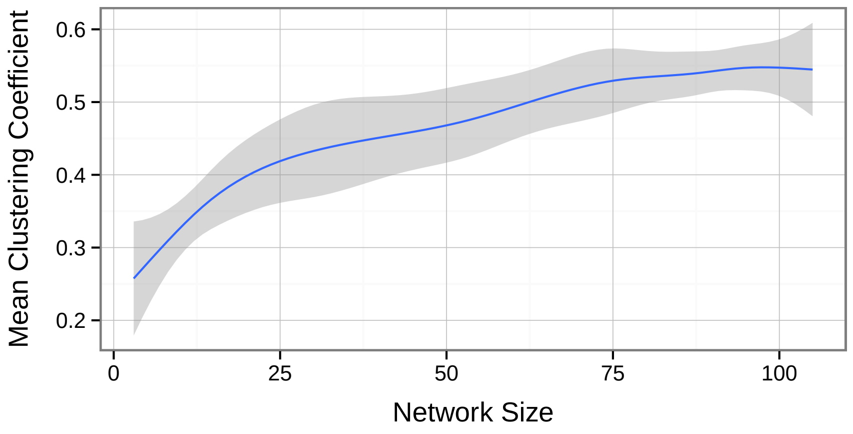


Figure 4.10: Clustering coefficient versus network size for the history of Django, with a light gray boundary to indicate the 99.5% confidence intervals.

It has been hypothesized that, at a critical upper bound, the cost incurred from the overhead of coordination exceeds the benefit of coordinating [Bro78]. Our results indicate that this bound indeed does exist and that developer coordination is constrained to evolve in a manner that promotes groups to form but not to exceed an upper bound. The evidence shown here is not definitive proof of a bound, but it is supportive of the conjecture that a bound exists. Thus, we *accept H2*.

4.3.5 Hierarchy

In Section 4.2.5, we introduced the concept of hierarchy in terms of its relation to scale freeness and modularity: hierarchy in complex networks is operationalized by a linear dependence between the log-transformed clustering coefficient and the node degree. We illustrate the results of applying the method described in Section 4.2.5 in Figure 4.11, where the evolution of hierarchy in an early stage (top) and late state (bottom) is shown for Firefox. In the early state, we are able to see that the network exhibits global hierarchy, because a linear model of the form $Y = \beta_0 + \beta_1 X$ describes the observed data, where X is the degree of a developer and Y is the clustering coefficient. In Section 4.2.5, we showed that a hierarchical network has the property that the log-transformed degree and log-transformed clustering coefficient exhibit a linear dependence. The results indicates that the linear model achieves a good fit, which is evident by an R^2 value of 0.894. Furthermore, the p value indicates that the linear model slope parameter β_1 is significantly different from zero, and so we can conclude that global hierarchy is present. In the late stage (bottom figure), the linear model no longer describes the *global* set of developers, instead it only describes the high degree nodes. In this case, we can conclude that a global hierarchy is no longer present and hierarchy predominantly exists in the high degree developer group.

The principal evolutionary trend with respect to hierarchy is the following: In early stages, developers are arranged in a global hierarchy. In later stages, a hybrid structure emerges, where only the core developers are hierarchically arranged, but the global hierarchy is no longer present. We observed that there is a smooth transition between the early and late stages shown for Firefox in Figure 4.11, which leads to a gradual deconstruction of the global hierarchy. The gradual deconstruction process is shown in Figure 4.12, where we illustrate the continuous evolution of hierarchy over the entire history of LLVM. Each sample represents the slope parameter β_1 of the linear model describing the hierarchy in the project at a single point in time. We see that hierarchy is most significant (largest negative slope) at the start and is progressively lost until virtually no global hierarchy is present (i.e., near zero slope) in the most recent revision. The results for all projects are shown in Table 4.2, where column "${\beta_1}_{early}$" represents the linear-model slope parameter at an early stage and column "${\beta_1}_{late}$" represents the slope parameter at a late stage. The early stage represents the earliest analysis window with more than five developers present, and the late stage represents the most up-to-date analysis window. We are able to see that in all projects except one (Qt 4), ${\beta_1}_{early} < {\beta_1}_{late}$ indicating that the hierarchy has diminished over time.

The results certainly suggest that, from a global perspective, developer hierarchy diminishes with time, but the mechanism responsible for the transformation is not obvious. To investigate this process further, we examined the high-degree nodes (i.e. highly central, and thus important developers) and found that they are hierarchically arranged at all times. Furthermore, the mechanism for decomposing the global hierarchy is established through the introduction of low-degree and mid-degree nodes, which do not obey the hierarchy established

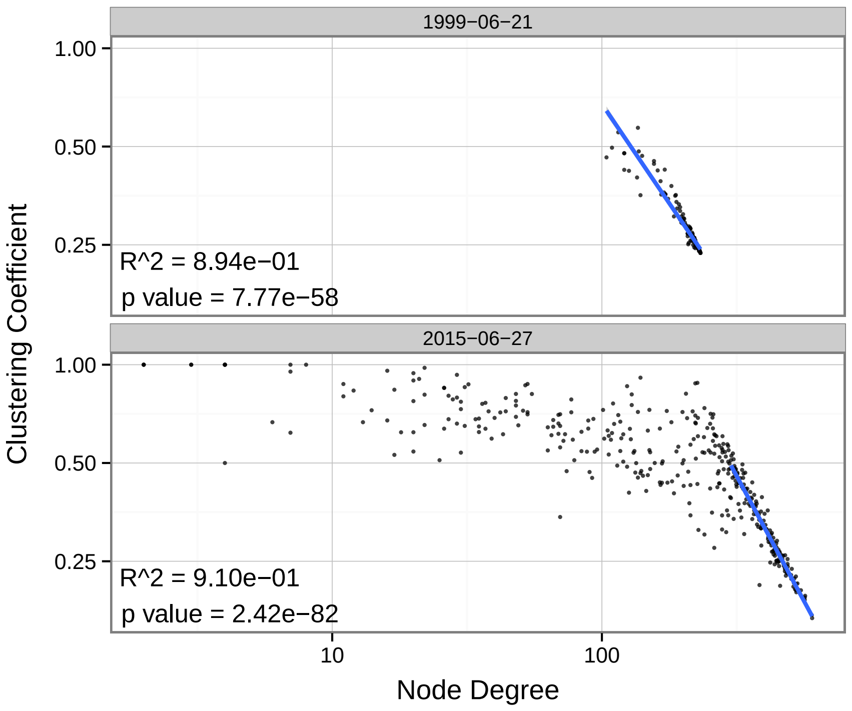


Figure 4.11: Early stage and late stage hierarchy of Firefox. The fitted linear model is superimposed on a scatter plot of node degree versus clustering coefficient. In the early stage (top), the linear model describes the complete data set indicating global hierarchy. In the late stage (bottom), global hierarchy is not present since only the high-degree nodes are described by the linear model. The linear model in the late stage has been fitted only to the high degree nodes.

by the high degree nodes. In essence, the developers become divided into two high-level organizational structures: The highest-degree nodes (developers with many coordination requirements) are hierarchically arranged and the mid-to-low-degree nodes (developers that are less central in the organizational structure) are not hierarchically arranged. This is visible in the late stage scatter plot of Figure 4.11, where beyond the break point (at a degree of roughly 250), the nodes obey a linear dependence and are thus hierarchically arranged. This evidence suggests that the differences between developers are not entirely explained by their distinct participation levels, but they also fundamentally differ in how they are structurally embedded in the organization.

As a project grows and becomes more complicated along numerous dimensions, we expected changes in the command-and-control structure. The expected trend was towards greater distribution of influence, which would

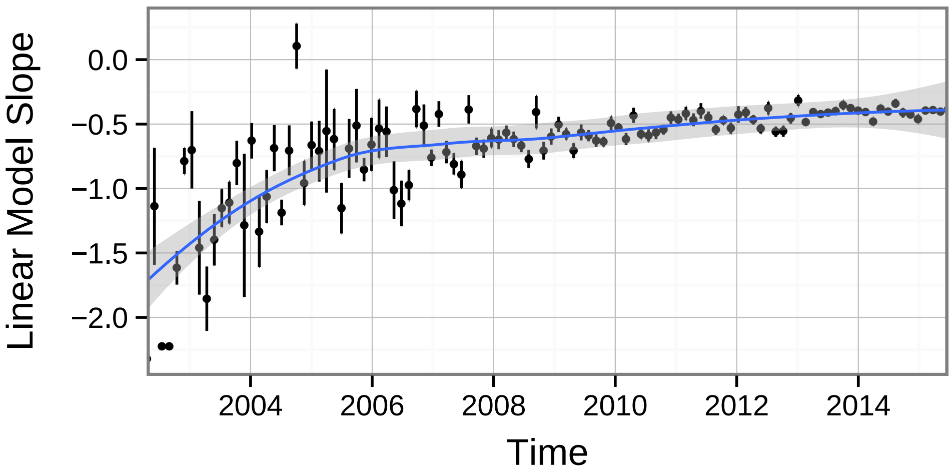


Figure 4.12: Evolution of hierarchy for the entire history of LLVM. The light gray boundary indicates the 99% confidence interval and error bars indicate the standard error on the slope estimate. The trend indicates that hierarchy is decreasing over time as the linear model slope β_1 tends towards zero.

manifest as the elimination of a global hierarchy. Our results indicate that, over time, hierarchy vanishes by the introduction of a large number of low and mid-degree developers that do not arrange into a hierarchy. We did also find evidence that extremely high-degree developers remain hierarchically arranged over time, though, they only constitute a very small faction of the project's developers. Overall, we *accept H3*, because the hypothesis is a statement regarding the global structure and, in that sense, the evidence indicates that global hierarchy vanishes over time.

4.4 Threats to Validity

External Validity We draw our conclusions from a manual selection of 18 open-source software projects. The manual selection and the choice to analyze only open-source software projects is a threat to external validity. We mitigated the consequences by choosing a wide variety of projects that differ in many dimensions and constitute a diverse population. Furthermore, we considered the entire history to prevent temporally biasing our results. We specifically chose only large projects with very active histories because our contributions are focused on understanding complexity in developer coordination, and in small projects (e.g., less than 10 developers), the coordination challenges are less severe. Due to certain limitations of our current infrastructure, we are unable to include complete analyses of software ecosystem projects, such as Eclipse[2], because they are typically distributed among multiple repositories.

Internal Validity We examined the evolution of developer networks over time, however, it is conceivable that factors other than time have an influence on the observed trends, threatening internal validity. By considering the influence of network size, we accounted for the most likely confounding factor. Furthermore, we found that the trends are often consistent across several projects, and we rigorously employed statistical methods to avoid drawing conclusions from insignificant fluctuations in the data.

Construct Validity Our methodology relies, to some extent, on the integrity of the data in the version-control system to generate a valid developer network, threatening construct validity. Since the version-control system is a critical element of the software-engineering process, it is unlikely that the data would be significantly corrupt. In terms of network construction, the heuristics we rely on have been shown to generate authentic developer networks, but do omit some edges [JMA$^+$15]. However, a few omitted edges would not severely impact the conclusions of our study. Furthermore, our enhancement of this form of developer networks to recover omitted edges is based on a technique that has been shown to authentically represent system coupling [BDO$^+$13]. The operationalizations of scale freeness, modularity, and hierarchy are thoroughly studied and well-established concepts in the area of network analysis. Although the application concepts from social network analysis to socio-technical developer networks is relatively new, empirical evidence is accumulating that suggests the metrics are reliable and valid [JMA$^+$15; MW11].

4.5 Discussion & Perspectives

The results of our study on the evolution of developer networks revealed several intriguing patterns. We will now discuss the relevance and potential explanation for these network patterns by linking them to software-engineering principles. Specifically, we discuss a likely model for growth of a project, a source of pressure for developers to become more coordinated with time, and the benefits of a hybrid organizational structure that is hierarchical for core developers and non-hierarchical for peripheral developers.

[2]`https://eclipse.org/`

RQ1: Change Our first research question asked what evolutionary adaptations are observable in the evolution of developer networks. In terms of scale freeness, we saw that projects do not begin in a scale-free state, instead this property emerges over time. Initially, the structure of developers exhibits high homogeneity and then, over time, hub nodes (i.e., very involved developers) appear that are responsible for a disproportionately large number of coordination requirements. We also saw that adaptations occur in the modularity of developer networks. Developers are loosely clustered initially, but, over time, clustering increases and gradually converges to a state where half of every developer's neighbors have coordination requirements. Finally, the structural property of hierarchy also changes over the course of time. Initially, projects have a globally hierarchical organization. Over time hierarchy is lost, as low degree developers are introduced to the network, which do not assimilate into the hierarchy. Still, hierarchy is always maintained for the highly-connected developers.

RQ2: Growth Our second research question focused on the relationship between these changes in the organizational structure and the scale of a project. Most of our subject projects experience steady growth over time. Presumably, many of the evolutionary principles we observed are closely related to the increasing scale of the project. For a couple of projects, we saw the growth stagnate or the overall size decrease. In these projects, we saw the reverse of what was seen during project growth. For example, the scale-free property was lost and clustering decreases. At this point, our results suggest a strong dependence between the scale of a project and the properties of its organizational structure. It appears that projects of different size exhibit different structural features of the organization. This result is interesting to the general software-engineering community, because it suggests that, when determining how to organize developers, it is crucial to consider how many developers will be involved in the implementation. In projects with few developers (e.g., less than 30), it may not be necessary to have developers that are highly dedicated to coordinating the work of others, and each developer can essentially occupy equivalent structural positions in the organization. However, in a very large project (e.g., 30 or more), it may be crucial to have developers entirely dedicated to coordinating the work of other developers and to occupy hub positions that span the organizational structure.

Scale freeness To better understand the growth behavior of developer networks, we examined the relationship between a project's growth state (increasing or decreasing) and the scale-freeness property of its developer network. The model of preferential attachment, which is the predominant generative model for scale-free networks, has the simultaneous requirements that the network must grow and that new nodes have a preference to attach to already well-connected nodes [BA99]. We found that, in this regard, the evolution of developer networks into a scale-free state is consistent with the model for preferential attachment. For several projects, the scale-freeness property is only observable during network growth and is lost during periods of growth stagnation or decrease, shown for Node.js in Figure 4.5. Furthermore, the loss of the scale-freeness property often precedes the stagnation or loss of developers. While it would be premature to make any strong statement about causality, the combination of correlation and preceding in time makes the loss of the scale-free state a conceivable predictor for the loss of growth in the project. These results suggest that, if a project grows beyond a certain size, the coordination structure will exhibit strong inhomogeneity in the distribution of coordination requirements among developers. It seems that there is a driving force that encourages a relatively small group of developers to bear the majority of the coordination burden. As a project achieves a large size (50 developers or more), the need for hub nodes in the coordination structure appears to be more critical. Software engineers should consider the project size when determining how to distribute the tasks among developers and how modes of collaboration between the multiple development sites should be realized.

Modularity In Section 4.3.4, we noted that an increasing clustering coefficient is a common evolutionary trend. This is a curious result because, in the ER random graph model (cf. Section 4.2.3), the clustering coefficient decreases with increasing network size, while in the preferential-attachment model, the clustering coefficient is independent of network size [RB03]. So, this result begs the question of what the driving force behind this unique evolutionary trend is. From the theory of software evolution, we expect that the natural tendency for an architecture is to become more strongly coupled over time, as complexity increases and initially clean abstraction layers deteriorate [LR01], which has been observed also in practice [MFD08]. Additionally, Conway's law suggests that the organizational structure and the structure of technical artifacts produced by the organization are constrained to mirror each other [Con68]. Based on these principles, we hypothesize that the evolution of the artifact structure is the driving force that influences developers to become more coordinated. Software engineers should be conscientious of the increasing demand on developers to coordinate with more developers as the software evolves. In the later stages of a projects, it may be critical to shift more attention towards mechanisms that support effective coordination between developers. It may even be necessary to reduce the task load on developers in later stages of a project, to ensure that the coordination requirements are given sufficient attention and resources; otherwise a decrease in software quality is a legitimate threat.

Hierarchy One of the most intriguing characteristics of open-source software projects is the strongly inhomogeneous distribution of effort between core and peripheral contributors [Koc04; MFH00; TMB10]. This characteristic is distinct from typical commercial development setups and is conceivably responsible for enabling open-source software projects to scale without reducing overall productivity, which violates conventional software-engineering wisdom [Bro78; Koc04]. Typically, core and peripheral developers are classified based on the number of commits, lines of code, or e-mails they contributed. Interestingly, we discovered that a dichotomy is also observable in the organizational structure, where one group of highly central developers is hierarchically organized, but the another group of less central developers is not hierarchically organized. We think that the reason for low degree developers not assimilating into the hierarchy stems from pressures to form a hybrid organizational structure that promotes regularity while also remaining flexible and robust to volatile developers. The process of software development demands a high degree of consistency and, for this reason, hierarchies are appropriate organizational structures. However, hierarchies are intrinsically inflexible structures [Kot14]. In open-source software projects, there is pressure for the organizational structure to remain flexible because, open-source software projects have high developer turnover rates for the peripheral developers, who constitute the majority of the contributors [OJ07]. It is conceivable that the existence of a hybrid organizational structure is even a signal of project health by indicating that the organization has responded to the adaptation pressures that are present in open-source software development. To the wider software-engineering community, this result indicates that a software project may benefit from embedding developers into the organizational structure differently depending on their experience level and likelihood of leaving the project. For example, a hierarchy can be an efficient structure when the members of the hierarchy a not likely to leave the organization. In the same way that open-source software development avoids embedding the volatile low degree developers into the hierarchy composed of highly central developers, it may be beneficial for any software project to avoid integrating inexperienced or potentially volatile developers into their hierarchical organizational structure.

4.6 Related Work

Lopez et al. first studied developer coordination by linking developers based on mutual contributions to modules for a static snapshot of three open-source software projects. They found that developer networks are not scale free, based on a visual inspection of the cumulative degree distribution [LRGH09]. Jermakovics et al. constructed networks based on contributions to files for three projects, and they developed a graph-visualization technique to represent the developer organizational structure [JSS11]. Toral et al. constructed developer communication networks based on the Linux kernel e-mail archives between 2001 and 2006 [TMB10]. They found that participation inequality is present in the communication network, and they introduced a core developer and peripheral-developer classification scheme. We differentiate our work by analyzing the entire project history and viewing developer coordination as an evolutionary process. Our network-construction procedure has demonstrated valid results with respect to capturing developers' perception of who they collaborate with and reveals a statistically significant community structure, which is obscured by the more coarse-grained approaches used in prior work [JMA+15]. Additionally, we use a fully automated and statistically rigorous framework to reduce subjectivity, and we draw our conclusions from 18 projects instead of just two or three. We build on prior work by explaining the commonly observed network features (e.g., participation inequality) in terms of the important structural concepts of scale freeness, modularity, and hierarchy.

Louridas et al. studied structural dependencies between classes and packages of 9 software systems using static source-code analysis techniques [LSV08]. They found that power-law distributions are a ubiquitous phenomenon in the dependency structure by fitting a line to the log-scale degree distribution. Our work is complementary by identifying power-law distributions in developers' coordination requirements. This is a step towards an empirical validation of Conway's law by showing that a necessary condition is met regarding the match between the organizational structure and technical artifact structure.

While there is a number of theories regarding developer turnover and its effects, current empirical results are limited. Foucault et al. examined the relationship between internal and external developer turnover on software quality in terms of bug density [FPB+15]. Consistent with current theories, they found that high external turnover has a negative influence on module-level bug density. Others have explored factors that contribute to developer turnover and motivations for long-term involvement [HPN10; SLW12; YBH12]. Mockus found that developers leaving the projects negatively influence code quality, while new developers entering the project have no influence [Moc10]. Oddly, the results of Mockus and Foucault et al. do not agree, which may suggest that the influence of turnover is dependent on additional context factors. In our work, we primarily focused on the relationship between the turnover characteristics of core and peripheral developer groups and how these distinct groups are structurally embedded in the organization. We use the distinct turnover rates to rationalize the evolution of the developer network as an optimization process.

Godfrey et al. were the first to study software evolution in open-source software and found that the Linux kernel violates principles of software evolution by achieving super-linear growth at the system level [GT00]. This was later supported by evidence extracted from the version control system of 8621 projects on SourceForge.net [Koc04]. Koch found that large open-source software projects violate several laws of software evolution established for commercial projects [Koc04]. Specifically, they showed that developer productivity is independent of the number of developers in the project—a direct violation of Brooks' law [Bro78]. Furthermore, participation inequality is common and increases with the system size—a result that we confirmed—but the increase in inequality does not influence developer productivity. Koch proposed that strict modularization, self-organization, and highly decentralized work are responsible for the high efficiency seen in open-source software projects, but this was never verified [Koc04]. In our study, we found that our more detailed methodology, which considers source-code structure and software coupling, supports prior observations. Furthermore, we were able to extend the body of knowledge by directly studying the evolution of coordination structures that are conjectured to be responsible for the remarkable properties of open-source software projects.

In a different, but nonetheless related field, researchers have shown that a network representing the collaborative organization of knowledge in Wikipedia, based on references between articles, is a scale-free network and the network growth is described by the law of preferential attachment [SL08]. By showing that power laws are present in open-source software project, we add to the evidence that healthy collaborative enterprises are ubiquitously scale-free.

4.7 Summary

In Chapter 3, we determined that developer networks constructed from fine-grained information are an authentic representation of the organizational structure of software developers that stems from coordination requirements. We found that fine-grained developer networks exhibit structure that is substantially more organized than uniform randomness and these highly organized structures exhibit statistical significance and real-world validity. In this Chapter, we extended our approach to generate a sequence of developer networks that capture the time-varying behavior of the network. We applied our approach to conduct a longitudinal empirical study of 18 substantial open-source software projects with respect to three fundamental organizational principles: scale freeness, modularity, and hierarchy. In this evolutionary study of developer networks we were able to identify several important insights that are concealed by the temporally static perspective adopted by prior studies.

Based on our study of 18 open-source software projects, we found that, in projects exceeding 50 developers, the coordination structure becomes scale free. The implication is that coordination is disproportionally concentrated around an extremely small faction of developers. We also found that developers tend to have an increasing number of coordination requirements with other developers, but the increasing trend is limited by an upper bound, where coordination requirements exist between roughly half of every developers neighbors. Additionally, we discovered that developers are hierarchically arranged in the early phases of a project, but in later phases the global hierarchy vanishes and a hybrid structure emerges, where highly central developers exist in the hierarchy and lower degree developers exist outside the hierarchy. Overall, the adaptations that we observed in the structural features balance the opposing constraints of supporting effective coordination and achieving robustness to developer withdrawal. From the results it is clear that significant structural changes occur in the coordination structure of a project over time, and particularly as developers are added.

In this chapter, we observed indicators of an organizational dichotomy with respect to the positioning of developers in the developer network. One group of developers is located in highly central positions, is involved in a disproportionately large number of coordination requirements, and is hierarchically organized. The other group tends to be less centrally located, bear very little of the total coordination burden, and are not hierarchically arranged. Prior researchers have observed a dichotomy in open-source software projects that corresponds to distinctly different roles that developers can fulfill, termed core and peripheral. Operationalizations of these roles are typically based on simple counts of individual developer activity such as the number of commits or lines of code contributed, but these operationalizations lack any relational information regarding developer–developer relationships. While the results in this chapter certainly encourage a deeper investigation into the organizational dichotomy, it is unclear if this dichotomy corresponds to developer roles. We also do not know how the organizationally defined dichotomy relates to existing operationalizations of developer roles, and more importantly, how well these operationalizations align with the perception of developers. In the following chapter, we will address these exact issues. We expect that the richer developer network, that explicitly captures developer–developer relations, should outperform the simpler representations in terms of reflecting meaningful divisions in developer roles. By demonstrating that developer networks are more expressive than simpler representations, we make important steps forward by providing practical justification for a network representation instead of the existing simpler non-network models.

CHAPTER 5

Classifying Developers into Core and Peripheral Roles

This chapter shares material with the ICSE'17 paper "Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics" [JAHM17].

In this chapter, we utilize developer networks as a basis for classifying developers into roles depending on their position and stability within the organizational structure. In Chapter 3, we introduced a fine-grained approach for constructing developer networks that exhibit real-world validity. We found that fine-grained developer networks display qualities of self-organizing systems in terms of higher-order structure manifested as developer communities. Building on this work in Chapter 4, we adapted our approach to generate a sequence of networks to capture the time-varying dynamics of developer networks. Again, we found several more indications that the network's evolution is defined by organizational principles that represent a significant departure from random behavior. In addition, we observed a dichotomy in the organizational principles where there appears to be two distinct developer groups with fundamentally different features. One group exhibits high centrality, hierarchical embedding, high positional stability, and is densely interconnected, while the second group exhibits low centrality, no hierarchical embedding, low positional stability, and is sparsely interconnected. What we will address in this chapter is whether we can derive practical value from developer networks by exploiting the aforementioned distinctive patterns to elicit meaningful insights about the different roles that developers play in the project. The insights that we elicit in this work shed light on how developer roles manifest in the mode of interaction among developers fulfilling the same or different roles, and in the developers' relative positioning and stability within the organizational structure.

5.1 Motivation and Contributions

Developer Roles In open-source software development, there are numerous roles that contributors adopt, each with distinct characteristics and responsibilities. The popular "onion" model—first proposed by Nakakoji et al. [NYN$^+$02]—comprises eight roles typically appearing in open-source software projects. These roles extend from passive users of the software, to testers and, active developers. According to this model, there is a clear and intentional expression of the substantial difference in scale between the group sizes fulfilling each role. That is to say, two groups fulfilling different roles will often differ in size by a significant margin. Quantitative evidence from several empirical studies substantiates this model by showing that the number of code contributions per developer is described by heavy-tailed distributions, which implies that a very small fraction of developers is responsible for performing the majority of work [CWLH06; DB05; MFH02]. The form of this distribution also implies that most developers contributing to a project make only few or irregular contributions. On the basis of these results, the distinction between different roles of developers is often coarsely represented as a dichotomy comprised of core and peripheral developers [CWLH06]. *Core developers* typically play an essential role in developing the system architecture and forming the general leadership structure, are characterized by prolonged, consistent, and intensive participation in the project, and they often have extensive knowledge of the system design and strong influence on project decisions [CWLH06; MFH02]. In contrast, *peripheral developers* are typically involved in bug fixes or small enhancements and are characterized by irregular, and often short-lived, participation in the project. The peripheral developer group is the larger of the two, by a significant margin, but core developers are responsible for doing most of the work [CH05; CWLH06; MFH02]. While peripheral developers are an abundant human resource, they also introduce risk and consume resources. For example, empirical evidence indicates that changes made by peripheral developers introduce more architectural complexity than changes made by core developers [TRC10]. Therefore, a stable and knowledgeable core developer group is imperative for ensuring system integrity in the presence of potentially inadequate changes introduced by peripheral developers. However, it appears to be ubiquitously true that successful open-source software projects are capable of benefiting from a large number of volatile peripheral developers, while at the same time mitigating the associated risks.

Role Stability All software projects face the situation that developers withdraw at some point and need to be replaced by new, often less experienced, developers. This process of *developer turnover* is known to present enormous risks to commercial projects, because crucial knowledge is often lost with departing developers [Boe89; Hus95; Moc10]. Researchers have shown evidence that developer turnover negatively impacts code quality, in terms of bug density [FPB$^+$15]. Another consequence of turnover is that replacement developers initially require mentorship, thereby consuming additional human resources by placing a burden on more experienced developers in the project. This is one factor that contributes to the well-known phenomenon that adding developers to a late project causes further delays [Bro78]. In open-source software projects, developer turnover exists in an extreme variation because the vast majority of developers have occasional, short-term participation, and generally only a very small number of core developers have consistent long-term participation [CH05; Koc04; MFH02]. It is extraordinary that open-source software projects are able to thrive under the extreme conditions of high developer turnover. For this reason, we dedicate attention to study the stability patterns of developer that fulfill different roles in the project.

Importance of Peripheral Developers At first glance, it seems that the larger group of peripheral developers represents an unnecessary threat to project success, as their volatile nature results in the known problems of knowledge loss and inadequate changes [TRC10]. However, there is evidence that supports an alternative story: peripheral developers are just as critical to the project's success as core developers [Ray99]. Without the peripheral group, there is limited opportunity for a vetting process to identify and promote appropriate developers [JS07]. This process is critical to establishing informed decisions regarding which developers are appropriate candidates for core positions. If a project wishes to remain robust to developer turnover and achieve sustainable growth, then there must be an adequate talent pool from which to draw new core developers. Furthermore, peripheral developers are crucial to the "many eyes" hypothesis—which posits that all bugs become shallow when the source code is scrutinized by a sufficiently large number of people—that is often referenced as an explanation for why open-source development will inevitable result in a high-quality product [Ray99]. Since core developers are in short supply, peripheral developers are the key for the project to benefiting from the consequences of many eyes.

Role Operationalizations Despite having a substantial understanding about the defining characteristics of core and peripheral developers and recognizing the importance of the interplay between these roles, there is significant uncertainty around core–peripheral operationalizations. A valid and reliable core–peripheral operationalization is crucial for testing empirical evidence of proposed theories regarding collaborative aspects of software development [HM03b; SSS07]. While several basic operationalizations have been proposed and loosely justified by abstract notions, they may be overly simplistic. For example, one common approach is to apply thresholding on the number of lines of code contributed by each developer [MFH02], but this could result in incorrectly classifying developers who just make large numbers of trivial cleanups. Further evidence suggests that, as a developer moves into a core role, their activity in terms of commit count or lines of code decreases substantially, because they shift their efforts to coordinating the work of others [JSW11]. The major weakness of existing core–peripheral operationalizations stems from the fact that they are primarily based on counting *individual developer* activity (e.g., lines of code, number of commits, number of e-mails sent), which lack any explicit consideration of *inter-developer* relationships. Since many of the defining characteristics developer roles are concerned with how they, or their actions, interact with other developers [JSW11; OSdO^{+}12], we see inter-developer relationships to be of primary importance for the operationalization of developer roles.

Contributions The contributions of this chapter can be summarized by two main achievements. Firstly, we statistically evaluate the agreement between the operationalizations of core and peripheral developers most commonly applied by researchers. We perform our study by examining data stored in the version-control systems (VCS) and developer mailing lists of 10 substantial open-source projects. The common operationalizations are termed “count-based” since they are simply based on counting individual developer activities, such as the number of commits or lines-of-code contributed. All count-based operationalizations claim to capture the same high-level concept which implies that if they are all valid operationalizations, they should reach consistent conclusions regarding the role of a given developer. Secondly, we establish and evaluate richer notions of developer role characteristics with a basis in relational abstraction. More specifically, we adopt a network-analytic perspective to explore manifestations of core and peripheral characteristics in the evolving organizational structure of software projects, as operationalized by fine-grained developer networks (cf. Chapter 3 & 4). For evaluation, we performed a survey among 166 developers to establish a ground-truth classification of developer roles to test whether the existing operationalizations and our network-based insights are consistent with respect to each other and valid with respect to developer perception. Our conjecture is that, if the abstract characteristics of core and peripheral developers proposed in the literature and supported by empirical evidence are accurate, these roles should also manifest in ways that transcend simple counts of individual developer contributions. In particular, we explore stability patterns and structural embeddings of core and peripheral developers in the global organizational structure of a project, which contains more actionable information regarding organizational or collaborative issues than just a count of code contributions.

In summary, we make the following contributions:

- We statistically evaluate the agreement between classifications of core and peripheral developers generated from commonly used operationalizations—henceforth called *count-based* operationalizations—by studying 10 substantial open-source projects, over at least one year of development, with data from two sources (version-control system and mailing list).

- We conducted a survey among 166 developers to establish a ground truth composed of 982 samples, which we use to evaluate the validity of several core–peripheral operationalizations with respect to developer perception.

- We identify structural and temporal patterns in the project's organizational structure that operationalize core–peripheral roles using network-analysis techniques, referred to as the *network-based* operationalizations.

- We demonstrate that network-based operationalizations exhibit moderate to substantial agreement with the existing count-based operationalizations, but the network-based operationalizations are more reflective of developer perception than the count-based operationalizations.

- We highlight and discuss a number of insights from our network-based operationalizations that are incapable of being expressed by count-based operationalizations, such as positional stability, hierarchical embeddings, and interaction patterns between core and peripheral developers.

5.2 Operationalizations of Developer Roles

We now introduce the count-based operationalizations, which attempt to differentiate core–peripheral roles by counting individual developer activities. This will be followed by the network-based operationalizations that capture core–peripheral roles by means of explicit modeling of inter-developer relationships.

5.2.1 Count-based Operationalizations

Based on a review of the existing literature, we have identified three variations of count-based operationalizations of core–peripheral roles [BGD$^+$07; CWLH06; dSFD05; MFH02; OSdO$^+$12; RGH09; RG06; TRC10]. In the literature, metrics are used with a quantile threshold to define a dichotomy composed of core and peripheral developers. We apply the standard 80th percentile threshold, because of its wide use and its justification based on the data following a Zipf distribution (cf. Section 5.6). Two operationalizations capture technical contributions to the version-control system and one captures social contributions to the developer mailing list.

Commit count is the number of commits a developer has authored (merged to the master branch). A commit represents a single unit of effort for making a logically related set of changes to the source code. Core developers typically make frequent contributions to the code base and should, in theory, achieve a higher commit count than peripheral developers.

Lines of code (LOC) count is the sum of added and deleted lines of code a developer has authored (merged to the master branch). Counting LOC, as it relates to developer roles, follows a similar rationale as commit count. As core developers are responsible for the majority of changes, they should reach higher LOC counts than peripheral developers. A potential source of error is that developers writing inefficient code or changing a large number of lines with trivial alterations (e.g., whitespace changes) could artificially affect the classification.

Mail count is the number of mails a developer contributed to the developer mailing list. Core developers often posses in-depth technical knowledge, and the mailing list is the primary public venue for this knowledge to be exchanged with others. Core developers offer their expertise in different forms: making recommendations for changes, discussing potential integration challenges, or providing comments on proposed changes from other developers. Typically, peripheral developers ask questions or ask for reviews on patches they propose. Core developers often participate more intensively and consistently and have greater responsibilities than peripheral developers, in general. This should result in core developers making a large number of contributions to the mailing list. This is still only a very basic metric because a developer answering many questions and one asking many questions will appear to be similar, and there is no inter-developer information, so who is speaking with whom or with how many people is completely ignored.

Each of the above metrics has a foundation rooted in our current empirical understanding of the characteristics of core and peripheral developers, but in the end, they are all relatively simple abstractions of a potentially multifaceted and complex concept [DB05]. A comparison between the resulting classification of developers from these different metrics will provide valuable insights into whether systematic errors exist in these count-based operationalizations, which we perform in Section 5.4.1. However, as the focus of these metrics is still only to assign developers exclusive membership to one of two unordered sets—without relational information between sets or within the sets—the insights offered by the classification are of limited practical value. To address this shortcoming, we propose a relational view on developer coordination and communication as the basis for developer-role operationalizations.

5.2.2 A Network Perspective on Developer Roles

A developer network is a relational abstraction that represents developers as nodes and relationships between developers as edges (cf. Section 2.4). The promise of a network perspective is greater practical insights concerning the organizational and collaborative relationships between developers [BPD$^+$08; CH13; CHC08; dSFD05; MW11]. But to what extent can this promise be fulfilled? So far, we know that developer networks, when carefully constructed on version-control-system (cf. Chapter 3) and mailing-list data [BPD$^+$08], can be both accurate in reflecting developer perception and reveal important functional substructure, or communities, with related tasks and goals. What can be elicited from developer networks regarding the core–peripheral dichotomy has not yet been greatly explored, especially in comparison to non-network-based approaches, which is our intention in this work. Practical opportunities for network insights are, for example: identifying core developers that are overwhelmed by the peripheral developers they need to coordinate with; structural equivalence (that is two nodes with the same neighbors) could reveal which core developers have similar knowledge or technical abilities, which helps to determine appropriate developers for sharing or shifting development tasks; structural holes between core developers may indicate deteriorating coordination; or a single globally central core developer may indicate an important organizational risk.

5.2.2.1 Developer Networks

We now present the details of our network-analytic approach for analyzing data from version-control systems and mailing lists to examine relational characteristics of core and peripheral developers. Intuition and prior research led us to the conclusion that the role a developer fulfills can change over time [JS07]. For this reason, we analyze multiple contiguous periods of each project over one year using overlapping analysis windows. We apply the same approach described in Section 4.2 for studying developer network evolution. Each analysis window is three months in length, and each subsequent analysis period is separated by two weeks (cf. Section 4.2.2). We chose three-month analysis windows, because it has been shown that, beyond this window size, the development community does not change significantly, but temporal resolution in their activities is lost [MW11].

E-mail networks The e-mail exchanges between members of an open-source project provide complementary information to the version-control system data that we have focused on in Chapter 3 and Chapter 4. Mailing list archives are particularly important in the operationalization of roles because core developers are often regarded as having a primary responsibility to coordinate the work of others, which requires them to communicate heavily on the mailing list (cf. Section 5.1 and Section 2.2.3). Furthermore, empirical studies have shown that, as a developer progresses towards a core role, their activity in the version-control system typically decreases because they shift to project management and leadership roles that results in reduced version-control system activity and increased mailing list activity [JSW11]. To obtain the e-mail correspondence for a given project, we download the mailing list archives either from *gmane* using *nntp-pull* or directly from the project's homepage to obtain an *mbox* formatted file containing all messages sent to the mailing list. Most projects have different mailing lists for different purposes. We consider only the primary mailing list for development-related discussions. We apply several preprocessing steps to remove duplicated messages, normalize author names, and organize the mails into threads using the *Message-IDs* and *In-Reply-To-IDs* (cf. Section 2.2.3). Furthermore, we decompose the *From* lines of each mail into a $\langle \text{name}, \text{e-mail address} \rangle$ pair. In some cases, only an e-mail address or only a name is possible to recover, and this can present issues with identifying all e-mails that a single person sent. To resolve multiple aliases to a single identity, we use a basic heuristic approach similar to the one proposed by Bird et al. [BGD$^+$06]. Despite the potential problems regarding author-name resolution—as developers accumulate valuable credibility through contributions to the mailing list—it is counterproductive for highly active individuals to use multiple aliases and conceal their identity. To construct a network representation of developer communication, we apply the standard approach, where edges are added between individuals who make subsequent contributions to a common thread of communication (cf. Section 2.4.2).

Version-control-system networks Data in version-control systems are organized in a tree structure composed of commits. We analyze only the main branch of development, as a linearized history, by flattening all branches merged to master. Furthermore, we analyze only the *authors* of commits, not the committer (which are expressed differently in Git), and attribute the commit to a unique individual using the same aliasing algorithm as for the mailing-list data. We count lines of code for each commit based on diff information, where the total line count is the sum of added and deleted lines. The network representation of developer activities in the version-control system is constructed using fine-grained *function*-level information, which was observed to produce authentic networks that agree with developer perception (cf. Chapter 3). In this approach, changes are localized to function blocks using source-code structure to identify when two developers edit interdependent lines of code. We enhance the network with semantic-coupling relationships between functions (cf. Section 4.2), which has shown to also reflect developer perception of artifact coupling [BDO$^+$13].

5.2.2.2 Network-based Operationalizations

Based on the known characteristics of core and peripheral developers (cf. Section 5.1), we expect that characteristics of core and peripheral developers manifest in ways that transcend the count-based operationalizations introduced in Section 5.2.1—an expectation that is also supported by a survey among 166 open-source developers (cf. Section 5.4.5).

Typically, metrics used to classify a developer as core or peripheral generally quantify the amount of participation a developer has in the project, such as lines of code or number of commits contributed [CH05; RGH09; TRC10]. A developer is then assigned to the core group if their level of participation is in the upper 20th percentile; all other developers are considered to be peripheral [RGH09; TRC10]. For our network-based operationalizations, we adopt a similar percentile defined threshold to classify the developers. Since our metrics are based on networks, an individual can appear in the network, yet be assigned a value of zero. A person can also fail to appear in the network because they make no contributions to the project at that time period. We differentiate between these cases by assigning developer to classes according to the following.

Core: developers with metric value in the upper 20th percentile

Peripheral: non-core developers with non-zero metric value

Isolated: developers with a metric value equal to zero

Absent: developers that did not participate

We consider developers with a metric value equal to zero to be distinct from the peripheral group since these developers tend have extremely low activity, often with only a single contribution. We want to prevent the isolated developers distinct character from distorting the analysis of the peripheral developer group, so we analyze them separately. Next, we introduce the five network-based operationalizations that are rooted in the structure and evolution of developer networks (cf. Section 5.2.2.1).

Degree centrality aims at measuring local importance. It represents the number of ties (edges) a developer has to other developers [BE05]. As essential members of the leadership and coordination structure, core developers associate with other core members and with peripheral developers that require their technical guidance. Peripheral developers are likely involved in only a small number of isolated changes and thus have only a limited number of interactions with other members of the development community. The expectation is that core developers then have a larger degree than peripheral developers.

Eigenvector centrality is a global centrality metric that represents the expected importance of a developer by either connecting to many developers or by connecting to developers that are themselves in globally central positions [BE05]. Since core developers are critical to the leadership and coordination structure, we expect them to occupy globally central positions in the developer network.

Hierarchy is present in networks that have nodes arranged in a layered structure, such that small cohesive groups are embedded within large, less cohesive groups (cf. Section 2.3.7). In a hierarchical network, nodes with high degree tend to have edges that span across cohesive groups, thereby lowering their clustering coefficient [RB03]. In Chapter 3, we presented results which demonstrated that developers tend to form cohesive communities, and we expect core developers to play a role in coordinating the effort of these communities of developers. We further found evidence in Chapter 4 that highly central developers are arranged hierarchically, but developers with low centrality are not. Based on this evidence, the core developers should have a high degree and low clustering coefficient, placing them in the upper region of the hierarchy, while peripheral developers should exhibit a comparatively low degree and high clustering coefficient, placing them in the lower region of the hierarchy.

Role stability is a temporal property of how developers switch between roles. As core developers typically attain their credibility through consistent involvement and often have accumulated knowledge in particular areas of the system over substantial time periods, we expect their stability in the developer network to be higher than for peripheral developers. Developer turnover—the process by which developers enter and withdraw from a project—provides important insights into the stability of the organizational structure of a project.

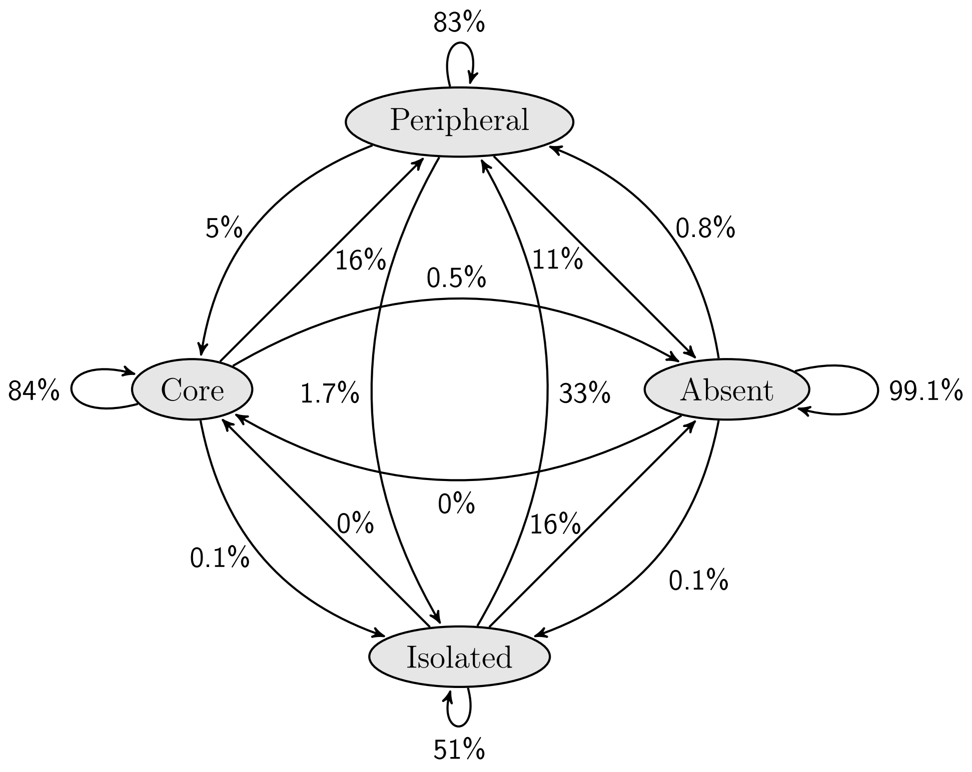
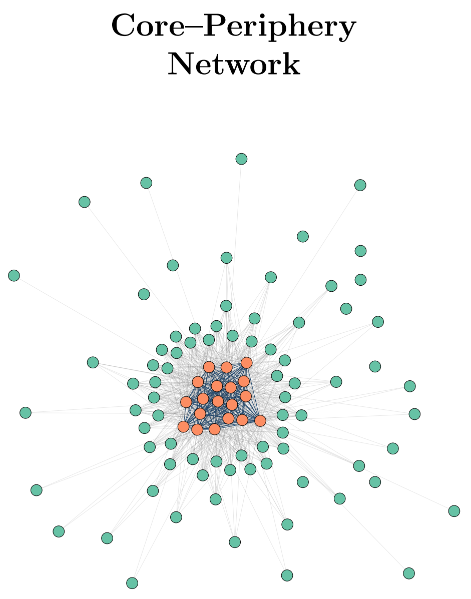


Figure 5.1: The developer-group stability for QEMU shown in the form of a Markov Chain. In some states, the addition of outgoing edge probabilities may not equal unity due to rounding errors.

We operationalize developer stability by estimating the probability that a developer in a given role leaves the project by not participating for, at least, 90 days. We also expand on this concept by not only studying the likelihood that developers leave a project but also the likelihood of transitioning between different roles in the project. Particularly, we employ sequential-data modeling techniques to formally address this concept. We make use of the discrete state Markov model by assigning a discrete state to every developer in the project for each time window (cf. Section 2.3.8). In the appendix (page 190), we discuss the trade-off involved in the choice of analysis windows and the influence it has on the Markov model. In this model, a developer is assigned to one of the four states defined above (core, peripheral, isolated, or absent) for each time window. To compute the transition probabilities, each developer's state transitions are expressed by a sequence of random variables $X_t \in \{s_1, s_2, s_3, s_4\}$ that can take on any of the four states. We then employ the Markov property such that $\Pr(X_{t+1} = x | X_1 = x_1, X_2 = x_2, \ldots, X_t = x_t) = \Pr(X_{t+1} = x | X_t = x_t)$. The assumption is that, to determine the next state transition, only information about the previous state is required. Using this assumption, we are able to represent developer transitions from state to state as an $N \times N$ transition matrix, in which each element indicates the probability of transitioning from any state in the state space N to any other state during the entire project's evolution. We used maximum-likelihood estimation to solve for each state transition parameter of the Markov model [Bis06]. We experimented with second order Markov chains, more formally $\Pr(X_{t+1} = x | X_t = x_t, X_{t-1} = x_{t-1})$, to test the validity of our assumptions, but the overall insights do not change and so we only show results for the simpler first order Markov chain.[1] Figure 5.1 provides an example developer transition Markov chain: The core developers stay in the core state in the following release with a 84% probability, transition to the peripheral state with 5% probability, with 0.1% probability transition to the isolated state, and with 0.5% probability to the absent state. All transition probabilities are between 0 and 1, and the sum of all transitions from a single state is equal to 1, to ensure that the conditions for a probability function are maintained.

Core–peripheral block model is a formalization, proposed in the social-network literature, that captures the notion of core–periphery structure based on an adjacency-matrix representation. A network exhibiting core–periphery structure and the corresponding block model is shown in Figure 5.2. The block model specifies the core–core region of the matrix as a 1-block (i.e., completely connected) shown in orange, the core–peripheral regions as imperfect 1-blocks shown in green, and the peripheral–peripheral region as a 0-block shown in white [ZMN15]. Intuitively, this model describes a network as a set of core nodes, with many edges linking each other, surrounded by a loosely connected set of peripheral nodes that have no edges connecting each other. Of course, this idealized block model is rarely observed in empirical data [BE00]. Still, we are able to draw practical consequences from this formalization by estimating the edge presence probability of each position to test if core and peripheral developers (operationalized by degree centrality) occupy core and peripheral network positions according to this block model. From the block model, one can mathematically reason that the probability of observing an edge in each block is distinct and related according to $p_{\text{core–core}} > p_{\text{core–periph}} > p_{\text{periph–periph}}$ [ZMN15]. This model aligns with empirical data that indicate that core developers are typically well-coordinated and are expected to be densely connected in the developer network [MFH02]. Since peripheral developers often rely on the

[1]The second order Markov chain is more complex by including the random variable X_{t-1} in the model, but the vast majority of variance for our data is explained by the first order Markov chain. We concluded that the increase in model complexity is not justified by the improvement in the model's fit.



Core–Periphery Block-model

Id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	■	1	1	1	0	0	0	1	0	0
1	1	■	1	1	0	0	0	0	0	1
2	1	1	■	1	0	1	0	0	0	0
3	1	1	1	■	1	0	1	0	1	0
4	0	0	0	1	■	0	0	0	0	0
5	0	0	1	0	0	■	0	0	0	0
6	0	0	0	1	0	0	■	0	0	0
7	1	0	0	0	0	0	0	■	0	0
8	0	0	0	1	0	0	0	0	■	0
9	0	1	0	0	0	0	0	0	0	■

Figure 5.2: (left) Developer network exhibiting core–periphery structure with the characteristic densely interconnected core group (orange nodes) and loosely coupled periphery group (green nodes). (right) Ideal block model of core–periphery structure, rows and columns are colored according to which group each block in the matrix corresponds to in the developer network. Not all nodes in the network are represented in the block model due to spatial constraints.

knowledge and support of core developers to complete their tasks, it follows that peripheral developers often coordinate with core developers, and only in rare cases would we expect substantial coordination between peripheral developers. This expected behavior aligns very well to the formalized notion of core–periphery positions from social-network analysis.

5.3 Empirical Study

We now present the details of our empirical study to test for agreement between the different count-based operationalizations of core and peripheral developer roles and to identify richer relational characteristics of these roles represented by our proposed network-based operationalizations. By means of a developer survey, we then evaluate each operationalization on the basis of how well it reflects developer perception.

5.3.1 Subject Projects

We selected ten open-source projects, listed in Table 5.1, to study the core–peripheral developer roles. We intentionally chose a diverse set of projects to avoid biasing the results. The projects vary by the following dimensions: (a) size (source lines of code from 50KLOC to over 16 MLOC, number of developers from 15 to 1000), (b) age (days since first commit), (c) technology (programming language, libraries used), (d) application domain (operating system, development, productivity, etc.), (e) development process employed. Developers of the project referred to as Project X have requested that their project name remains anonymous.

Table 5.1: Overview of subject projects

Project	Domain	Lang	Devs	SLOC	Commits	Date
Project X	User	C/++, JS	826	10M	276K	2015/12/05
Django	Devel	Python	100	430K	41K	2015/12/06
FFmpeg	User	C	103	1M	78K	2015/11/08
GCC	Devel	C/++	122	7.5M	144K	2015/11/03
Linux	OS	C	1467	18M	637K	2015/12/05
LLVM	Devel	C/++	180	1.1M	62K	2015/11/02
PostgreSQL	Devel	C	17	1M	40K	2015/12/05
QEMU	OS	C	134	1M	43K	2015/11/02
U-Boot	Devel	C	142	1.3M	35K	2015/11/01
Wine	User	C	62	2.8M	110K	2015/11/06

5.3.2 Research Questions

While each of the approaches for classifying core and peripheral developers is inspired by common abstract notions rooted in empirical results, it has not been shown that the approaches agree. It may be the case that they capture orthogonal dimensions of the same abstract concept, which gives rise to our first research question:

RQ1: Consistency—*Do the commonly applied operationalizations of core and peripheral developers based on version-control-system and mailing-list data agree with each other?*

Compared to the extent of our knowledge regarding the characteristics of core and peripheral developers, existing count-based operationalizations are relatively simple. Since core developers often have strong ownership over particular files and play a central role in coordinating the work of others on those artifacts [CHC08; JSW11; MFH02], we would expect core developers to differ, in a relational sense, from peripheral developers in how they are embedded in the communication and coordination structure. Furthermore, as core developers typically achieve their status through long-term and consistent involvement [JS07], we expect their temporal stability patterns to differ from peripheral developers.

RQ2: Positions & Stability—*Do the differences between core and peripheral developers manifest in relational terms within the communication and coordination structure with respect to their positions and stability?*

The utility offered by an operationalization is limited by the extent to which the operationalization is able to accurately capture a real-world phenomenon. So far, it is unclear to what extent the core–peripheral operationalizations reflect developer roles as seen by their peers. We explore whether relational abstraction, as in the network-based operationalizations, offers improvements over the count-based operationalizations by more accurately reflecting developer perception through explicit modeling of developer–developer interactions.

RQ3: Developer Perception—*To what extent do the various count-based and network-based operationalizations agree with developer perception?*

5.3.3 Hypotheses

The existing count-based operationalizations of core and peripheral developers discussed in Section 5.2.1 claim to be valid measures, and if this is a matter of fact, we expect to reach consistent conclusions about whether a given developer is core or peripheral. Due to finite random sampling and sources of noise, we expect imperfect agreement between two operationalizations even if they are consistent in capturing the same abstract concept. However, if the operationalizations are consistent, the level of agreement should be significantly greater than the case of random assignment of developer roles. Our null model for zero agreement is the amount of agreement that results from two operationalizations that assign classes according to a Bernoulli process.[2] To operationalize agreement between two binary classifications (core or peripheral) of a given set of developers, we use Cohen's kappa, $\kappa = (p_o - p_e)/(1 - p_e)$, where p_o is the number of times the two classifications agree on a role of a developer, divided by the total number of developers and where p_e is the

[2]A Bernoulli process generates a sequence of binary-valued random variables that are independent and identically distributed according to a Bernoulli distribution. The process is essentially simulating repeated coin flipping.

expected probability of agreement when there is random assignment of roles to developers, but the proportion of each class is maintained. Cohen's kappa is more robust than simple percent agreement because it incorporates the effect of agreement that occurs by chance [LK77]. This characteristic is particularly important in our case since the frequency of roles is highly asymmetric as the majority of developers are peripheral and only a small fraction are core. The ranges for Cohen's kappa and corresponding strengths of agreement are: 0.81–1.00 almost perfect, 0.61–0.80 substantial, 0.41–0.6 moderate, 0.21–0.40 fair, 0.00–0.20 slight, and < 0.00 poor [LK77].

H1—*Existing count-based operationalizations of core and peripheral developers based on version-control-system and mailing-list data are statistically consistent in classifying developer roles.*

The characteristics of core and peripheral developers discussed in Section 5.1 draws attention to the multitude of facets in which the two groups differ (e.g., contribution patterns, knowledge, level of engagement, organization, responsibility, etc.). While existing operationalizations of core and peripheral developers are primarily based on simple metrics of counting high-level activities of developers, these metrics largely ignore the richness in the definition of core and peripheral roles. In particular, the dimension of time is largely ignored, though time plays a central role in the developer-advancement process [JS07]. For example, over time, developers withdraw from projects and there are potentially severe consequences as a result of the loss of knowledge and the additional resources required to mentor replacement developers [Moc10]. However, many successful open-source software projects have adapted to benefit from an abundant supply of inherently unstable developers by ensuring that a more stable developer group exists and plays a substantial role in coordinating the efforts the more unstable developers [CH05; JS07; JSW11; MFH02]. Therefore, we expect that turnover characteristics to be of fundamental importance for determining developer roles. Additionally, in the count-based metrics, the relative positioning of developer in the organizational structure are ignored. But a difference in how core and peripheral developers are embedded in the organizational structure is to be expected, since core developers have extensive involvement in the coordination of specific technical artifacts and preside over peripheral developers. Therefore, we expect to see manifestations of the differences between the two distinct groups of developers in the developer network.

H2—*The well-known abstract characteristics of core developers will manifest as distinct structural and temporal features in the corresponding developer network:*

Core developers will exhibit globally central positions, relatively high positional stability, and hierarchical embedding.

As core developers form the primary coordination structure, we expect to observe: many edges in the developer network between core developers, less edges between core and peripheral developers, and even fewer edges between peripheral developers. We investigate this hypothesis in terms of preferences between the groups to associate based on the probability of an edge occurring between them according to the core–peripheral block model (cf. Section 5.2.2.2).

H3—*Core developers have a preference to coordinate with other core developers; peripheral developers have a preference to coordinate with core developers instead of other peripheral developers.*

We expect developer networks to reveal core and peripheral developers, albeit in a richer representation, with comparable precision to the currently accepted operationalizations. More specifically, we expect developer networks capture the core–peripheral property to an equally high standard as the currently accepted operationalizations; any disagreement should be on the order of the discrepancy between existing operationalizations.

H4—*The core–peripheral decomposition obtained from developer networks will be consistent with the core–peripheral decomposition obtained from the prior accepted operationalizations. The discrepancy in agreement will not exceed the amount observed between the existing operationalizations.*

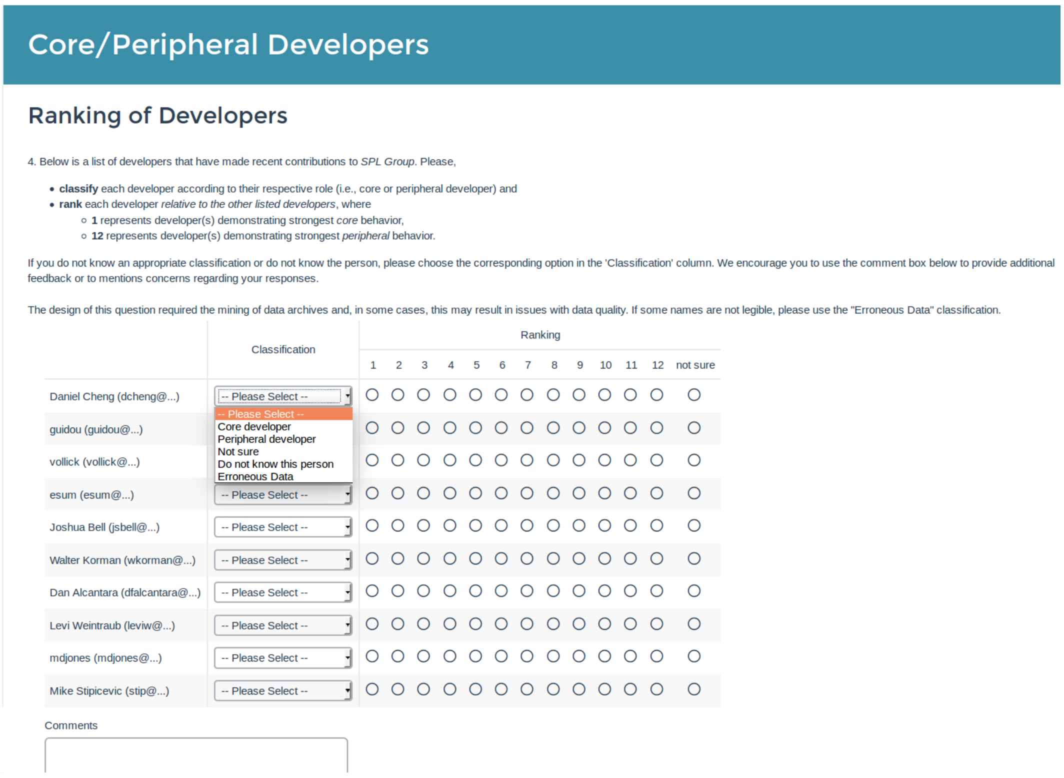
As the count-based operationalizations appear to reasonably capture simple aspects of developer roles, we expect a certain level of agreement between these operationalizations and developer perception. In the case of the network-based operationalizations, we expect even higher agreement with developer perception since the relational abstraction explicitly captures developer–developer interactions, which are neglected by the count-based operationalizations.

H5—*Count-based operationalizations agree with developer perception, but network-based operationalizations exhibit higher agreement.*

5.3.4 Developer Perception

To establish a ground-truth classification of developer roles, we designed an online survey in which we asked developers to report the roles of developer's in their project according to their perception. The goal of acquiring these data is to test whether the core–peripheral operationalizations are valid with regard to developer perception (not only to other operationalizations). A sample of the survey instrument can be found at the supplementary Web site.

We recruited participants for the study from the version-control-system data of our ten subject projects by identifying the e-mail addresses of individuals that made a commit within the three months prior to the survey date (cf. Table 5.1). This was to ensure that the selected developers have current knowledge of the project state, so that their answers are temporally consistent with our analysis time frame. One subject project, GCC, was excluded from the survey because the developer e-mail addresses are not available in the version-control system. For the remaining 9 projects, we sent recruitment e-mails to 3369 developers of which 166 elicited a complete response. In total, we obtained 982 role classifications. The distribution of responses from the projects was 41% for Linux, 7% for Django, 8% for QEMU, 15% for LLVM, 8% for PostgreSQL, 13% for Wine, and 7% for FFmpeg.


Core/Peripheral Developers

Ranking of Developers

4. Below is a list of developers that have made recent contributions to *SPL Group*. Please,

- **classify** each developer according to their respective role (i.e., core or peripheral developer) and
- **rank** each developer *relative to the other listed developers*, where
  - **1** represents developer(s) demonstrating strongest *core* behavior,
  - **12** represents developer(s) demonstrating strongest *peripheral* behavior.

If you do not know an appropriate classification or do not know the person, please choose the corresponding option in the 'Classification' column. We encourage you to use the comment box below to provide additional feedback or to mentions concerns regarding your responses.

The design of this question required the mining of data archives and, in some cases, this may result in issues with data quality. If some names are not legible, please use the "Erroneous Data" classification.

	Classification	Ranking												
		1	2	3	4	5	6	7	8	9	10	11	12	not sure
Daniel Cheng (dcheng@...)	-- Please Select --	○	○	○	○	○	○	○	○	○	○	○	○	○
guidou (guidou@...)		○	○	○	○	○	○	○	○	○	○	○	○	○
vollick (vollick@...)		○	○	○	○	○	○	○	○	○	○	○	○	○
esum (esum@...)	-- Please Select --	○	○	○	○	○	○	○	○	○	○	○	○	○
Joshua Bell (jsbell@...)	-- Please Select --	○	○	○	○	○	○	○	○	○	○	○	○	○
Walter Korman (wkorman@...)	-- Please Select --	○	○	○	○	○	○	○	○	○	○	○	○	○
Dan Alcantara (dfalcantara@...)	-- Please Select --	○	○	○	○	○	○	○	○	○	○	○	○	○
Levi Weintraub (leviw@...)	-- Please Select --	○	○	○	○	○	○	○	○	○	○	○	○	○
mdjones (mdjones@...)	-- Please Select --	○	○	○	○	○	○	○	○	○	○	○	○	○
Mike Stipicevic (stip@...)	-- Please Select --	○	○	○	○	○	○	○	○	○	○	○	○	○

Comments

Figure 5.3: Example survey question. Participants were instructed to classify each developer and rank them relative to each other based on the role the developer fulfills.

The survey includes two primary sections: The first section contains questions that require the developers to self-report their role in the project (core or peripheral) and to provide a textual description of the nature of their participation. This question is useful for identifying potential sampling-bias problems and to determine if developers' self-reported role is consistent with the answers provided by their peers. The second section includes a list of 12 developers, identified by name and e-mail address, sampled from their specific project. An example of this question is provided in Figure 5.3 with a shortened developer list (10 names instead of 12) to save space. For each developer appearing in the list, the respondent was asked to provide a classification of the developer's role. Appropriate options are also available if the respondent did not know the developer in question or was unsure of the role. We applied the following sampling strategy to select the list of twelve developers: Five developers were randomly selected from the core group and five from the peripheral group, classified according the the commit count-based operationalization (cf. Section 5.2.1). The remaining two developers were randomly selected from the direct neighbors, in the developer network, of the survey participant. We chose to use neighbors because it is likely that neighbors work directly together and are aware of each other's roles.

5.4 Results

We now present the results of our empirical study and address the five hypotheses stated in Section 5.3.3. For practical reasons, we are only able to present figures for a single project that is representative of the general results. Please refer to the supplementary Web site for the remaining project figures.

5.4.1 RQ1: Consistency of Count-Based Operationalizations

To address H1, we compute the pairwise agreement between all count-based metrics for a given project. For this purpose, we analyze each subject project in a time-resolved manner using a sliding-window approach (cf. Section 4.2.2) to generate time-series data that reflect the agreement for a particular three-month development period. An example time series is shown in Figure 5.4 for QEMU. While being only one project, the insights are consistent with the results from the other projects. The figure illustrates the agreement for Cohen's kappa, and we see that, for all comparisons, the agreement is greater than fair (e.g., greater than 0.2), which significantly exceeds the level of agreement expected by chance (cf. Section 5.3.3). This is evidence that the different count-based operationalizations do not contradict each other. For operationalizations that are based on the same data source (i.e., version-control system), we typically see substantial agreement (0.61–0.8). One reason for the

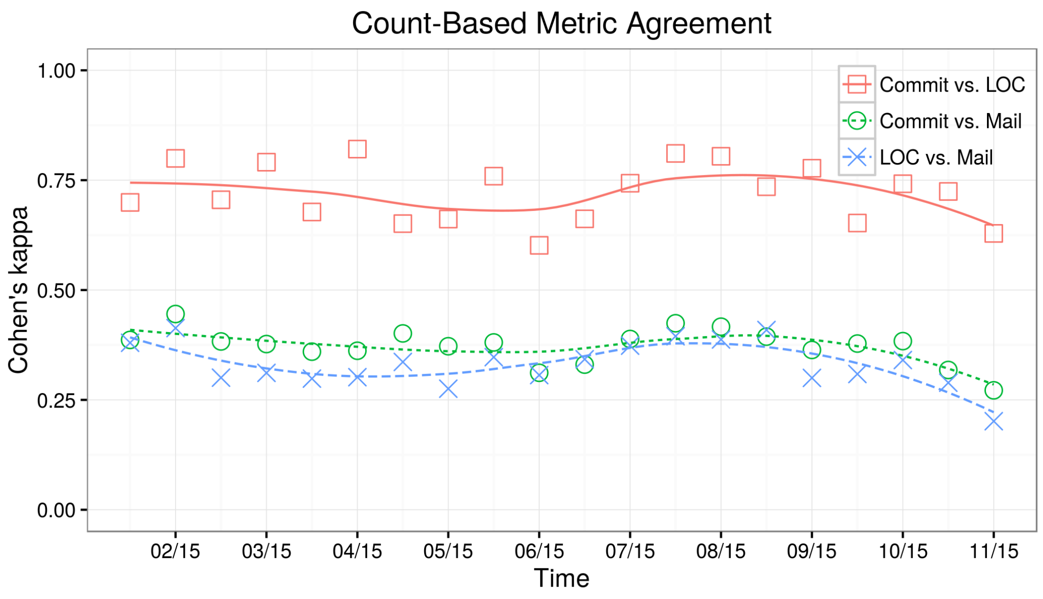


Figure 5.4: QEMU time series representation of pairwise agreement between count-based operationalizations. The data indicate that agreement is fair to substantial and is temporally stable (i.e., mean and variance are time invariant)

.

lower cross-archive agreement could be due to problems of multiple aliases, which will be discussed in detail in Section 5.5. Another interesting result is that the agreement is relatively stable over time, which is again visible in Figure 5.4 for QEMU. More specifically, the arithmetic mean and variance do not significantly change over time—a property referred to as “wide-sense stationary” in the time-series analysis literature [Ham94]. This feature of the data is a testament to the validity of the operationalizations, as we would not expect the agreement between operationalizations to change drastically from one development window to the next. The wide-sense stationary property is also important because it permits us to aggregate the data by averaging over the time windows to attenuate noise and generate more concise overviews without sacrificing scientific rigor or interpretability of the result.

Overall, the results demonstrate that the count-based operationalizations largely produce consistent results regarding the classification of developers into core and peripheral groups. We therefore *accept H1*.

5.4.2 RQ2: Core and Peripheral Manifestations in Developer Networks

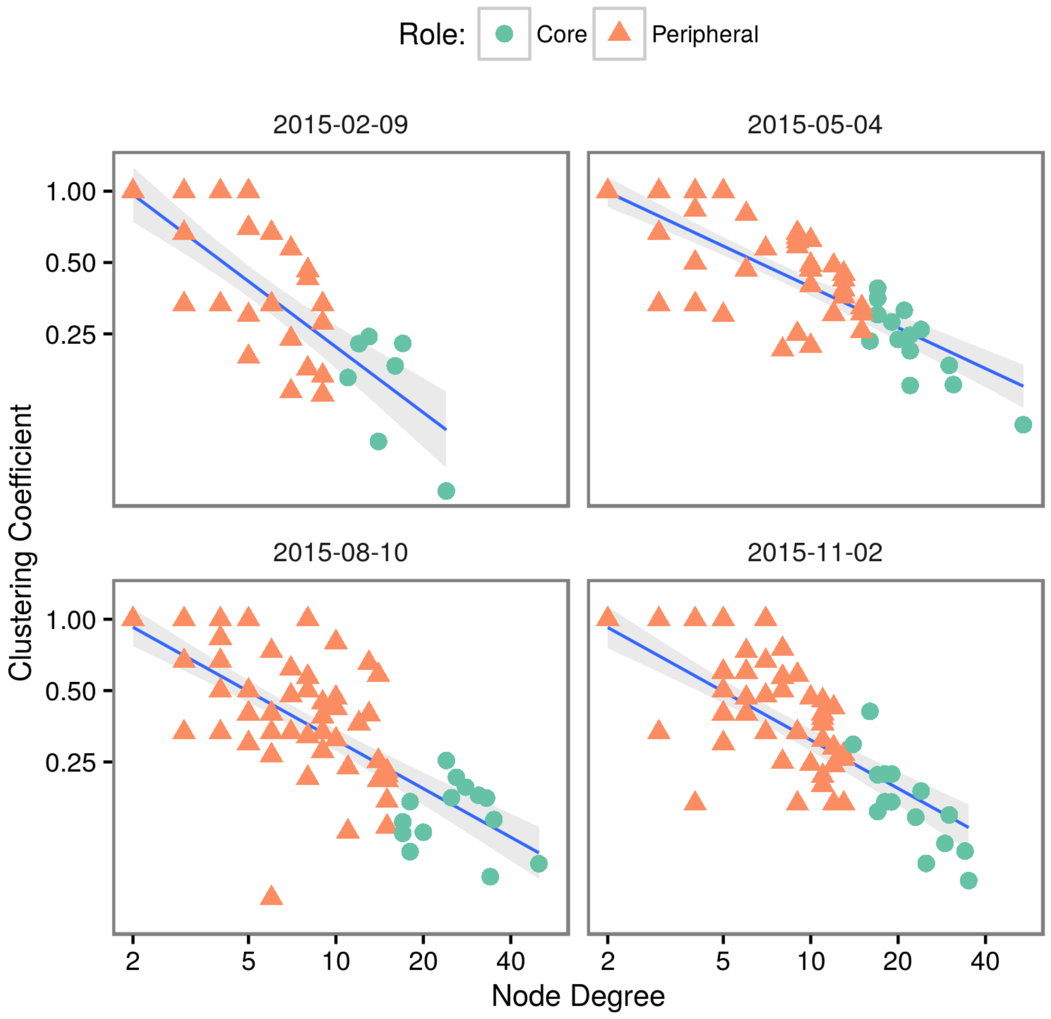


Figure 5.5: QEMU hierarchy during four development periods. The linear dependence between clustering coefficient and degree expresses the hierarchy. Core developers should appear clustered at the top of the hierarchy (bottom right region) and peripheral developers at the bottom of the hierarchy (upper left region)

Hierarchy In a hierarchical network, nodes at the top of the hierarchy have a high degree and low clustering coefficient; nodes at the bottom of the hierarchy have a low degree and high clustering coefficient (cf. Section 2.3.7). If hierarchy exists in a developer network, we should see mutual dependence between the clustering coefficient and the degree of nodes in the network that serves to separate the core group from the peripheral group. The hierarchical relationship for QEMU is shown in Figure 5.5; there is an dependence between the node degree and clustering coefficient and there is an obvious separation between the core and peripheral groups. Nodes with a high degree are seen to exclusively have very low clustering coefficient and are indicative of core developers according to Section 5.2.2.2; low degree nodes have consistently higher clustering coefficients and are indicative of peripheral developers. For the remaining subject projects we present the dependence in a more compact form in terms of Spearman's correlation coefficient between clustering coefficient and degree (cf. Table 5.2 "Hierarchy"). We see that, for all projects, there is a strong negative correlation, which suggests that core and peripheral roles manifest in the hierarchical embedding of developers in the coordination structure. In Section 5.4.4, we will evaluate whether developer perception confirms or rejects the hypothesis that a developer's relative position in the hierarchy is an organizational manifestation of their particular role.

Table 5.2: Results for block-model edge probabilities and hierarchy

	Edge Probabilities			Hierarchy	
Project	C–C	C–P	P–P	Rho[1]	p value
Project X	9.75e-02	4.19e-03	2.70e-03	-0.552	5.51e-33
Django	2.95e-01	9.09e-03	3.08e-03	-0.812	1.28e-06
FFmpeg	5.50e-01	2.44e-02	5.16e-03	-0.725	7.10e-06
GCC	4.07e-01	1.84e-02	1.01e-02	-0.646	1.12e-04
Linux	2.39e-02	5.93e-04	3.60e-04	-0.689	6.06e-62
LLVM	7.80e-01	5.54e-02	2.62e-02	-0.778	8.72e-24
PostgreSQL	1.00e+00	1.62e-01	5.13e-02	-0.871	1.31e-03
QEMU	3.20e-01	1.95e-02	1.16e-02	-0.756	4.76e-07
U-Boot	2.00e-01	7.59e-03	4.20e-03	-0.728	8.27e-05
Wine	3.46e-01	2.91e-02	1.28e-02	-0.832	1.04e-05

[1] Spearman's correlation coefficient

Stability Developers who fulfill a particular role within a project and who maintain participation over subsequent development periods are defined to be stable (cf. Section 5.2.2.2). We study this characteristic by examining the developers' transitions from one state to another (e.g., core to peripheral) in a time-resolved manner. The result of examining the developer transitions over one year of development for QEMU are shown in Figure 5.6. In this figure, the transition probabilities between developer states are shown in the form of a Markov chain. We applied the procedure described in Section 5.2.2.2 to construct a Markov chain representing the transitions between the four possible developer states (core, peripheral, isolated, and absent). The Markov chain for QEMU will be described as a representative of the primary result. For developers in the "core" state, we see they are very unlikely to leave the project with only a 0.5% chance of occurring. In comparison to developers in the peripheral state and isolated state, the chance of becoming absent is 11% and 16% respectively. We see that it is a common result that the core developers are 5 to 10 times less likely to leave a project in comparison to peripheral developers. This result is convincing evidence that the groups of peripheral and isolated developers, or more generally non-core developers, are significantly less stable than core developers. Furthermore, we see that, once a developer enters the absent state, there is an overwhelming probability that the developer will not return to the project. This result suggests, once a developer becomes absent for a single revision, in most cases, she will not participate in contributing code in the future. Since entering the absent state most likely indicates a total lost of the individual and any valuable knowledge they possess, the peripheral developers are a potential source of risk because of their high volatility. The overall observation is that developers in a core state are substantially less likely to transition to the absent state (i.e., leave the project) or isolated state (i.e., have no neighbors in the developer network by working exclusively on isolated tasks), in comparison to developers in a peripheral state. Based on this result, the core developers represent a more stable group than peripheral developers.

Core–periphery block model The core–periphery block model describes the core and peripheral groups, formalized as positions in a network, as a particular two-class partitioning of nodes (cf. Section 5.2.2.2). To test whether our empirical data are plausibly described by the core–periphery block model, we must compute the edge-presence probabilities for core–core, core–peripheral, and peripheral–peripheral edges. If the edge-presence probabilities are arranged according to, $p_{\text{core–core}} > p_{\text{core–periph}} > p_{\text{periph–periph}}$, then we can conclude that core developers constitute the most coordinated developers in the project, peripheral developers coordinate primarily with core developers, and peripheral developers rarely coordinate with other peripheral developers. This provides an example of a relational perspective that captures intra- and inter-relational information on developer roles (cf. Section 5.2.2.2).

The edge-presence probabilities for all projects are shown in Table 5.2 (column "Edge Probabilities"). In all projects, the inequality holds, indicating that the model plausibly describes our projects. The edge-presence probability for core–core has a mean value of 4.02×10^{-1}, for core–peripheral edges it is significantly lower with a mean value of 3.30×10^{-2}, and the peripheral–

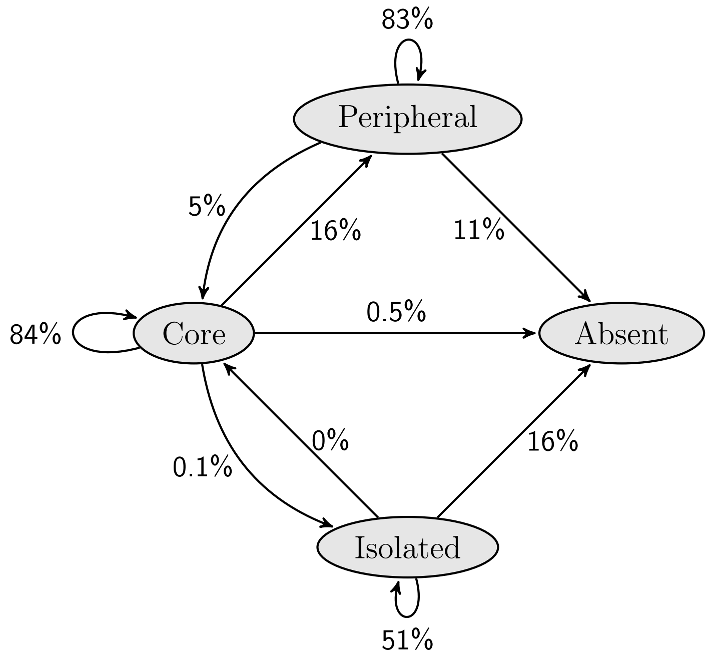


Figure 5.6: Developer-group stability for QEMU shown in the form of a Markov Chain. A few less important edges have been omitted for visual clarity.

peripheral edge probability is lower yet with a mean value of 1.28×10^{-2}. The interpretation is that peripheral developers are twice as likely to coordinate with core developers as opposed to other peripheral developers.

Two projects are noteworthy outliers, but are still described by the core–periphery block model: Linux and PostgreSQL. For Linux, the edge-presence probabilities are notably lower in all cases, and the difference in scale between core–core edge probabilities and the others is two orders of magnitude. In the case of PostgresSQL, we see an outlier in the opposite direction. The core–core edge probability is 1, notably higher than for all other projects, much like core–peripheral edges. It is interesting that both of these projects are also outliers in terms of the size of the developer community: Linux is much larger than most projects (1510 developers), PostgreSQL is much smaller (18 developers). From this result, it appears that the scale of a project influences how likely it is for developers to coordinate, and this influence has a greater effect on the coordination of peripheral developers.

Overall, the network-based operationalizations illustrate clear manifestations of core and peripheral developer roles that are consistent with known characteristics of these roles established by earlier empirical work. We also found evidence in terms of the core–peripheral block model that developer roles imply specific coordination preferences. On the basis of these results, we *accept H2 and H3.*

5.4.3 Agreement: Network-Based vs. Count-Based

So far, our results have provided evidence that the count-based operationalizations produce consistent classifications of developers, which is a testament to their validity, and that developer networks exhibit specific characteristics that are indicative of core and peripheral developer roles. Next, we present the results to relate the network-based to the count-based operationalizations of core and peripheral developers. We approach this evaluation again using Cohen's kappa by averaging the level of agreement over one year of development. QEMU is used as an example project and the pairwise agreement for each operationalization is illustrated in Figure 5.7. The stability and core–periphery block-model operationalizations do not show up explicitly since they are derived from degree centrality.

In general, the level of agreement always exceeds 0, which indicates that the strength of agreement between all operationalizations significantly exceeds what is expected by chance. The rows/columns beginning with "VCS" are based on data stemming from the version-control system, and those with "E-mail" are based on the mailing list. We again see that agreement between operationalizations defined on same data source typically have substantial agreement (0.6–0.8).

Overall, the results indicate that the network-based and count-based operationalizations are mostly consistent. While the agreement is imperfect, the results show that the divergence from perfect agreement is similar what is seen among the count-based operationalizations. We therefore *accept H4.*

5.4.4 RQ3: Developer Perception vs. Network-Based and Count-Based Operationalization

To establish a ground-truth classification of developers based on the perception of our survey participants, we computed the number of core and peripheral votes for each developer from the survey responses (cf. Section 5.3.4). For each developer, we assigned the ground truth role by determining which role was given the highest number of votes and, if the vote count for core and peripheral was equal, the developer was removed from the study. Upon inspection of the responses, we found that they were largely consistent regarding a given developer's role. The condition of having equal number of votes for core and peripheral was extremely rare. The results of comparing both the count and

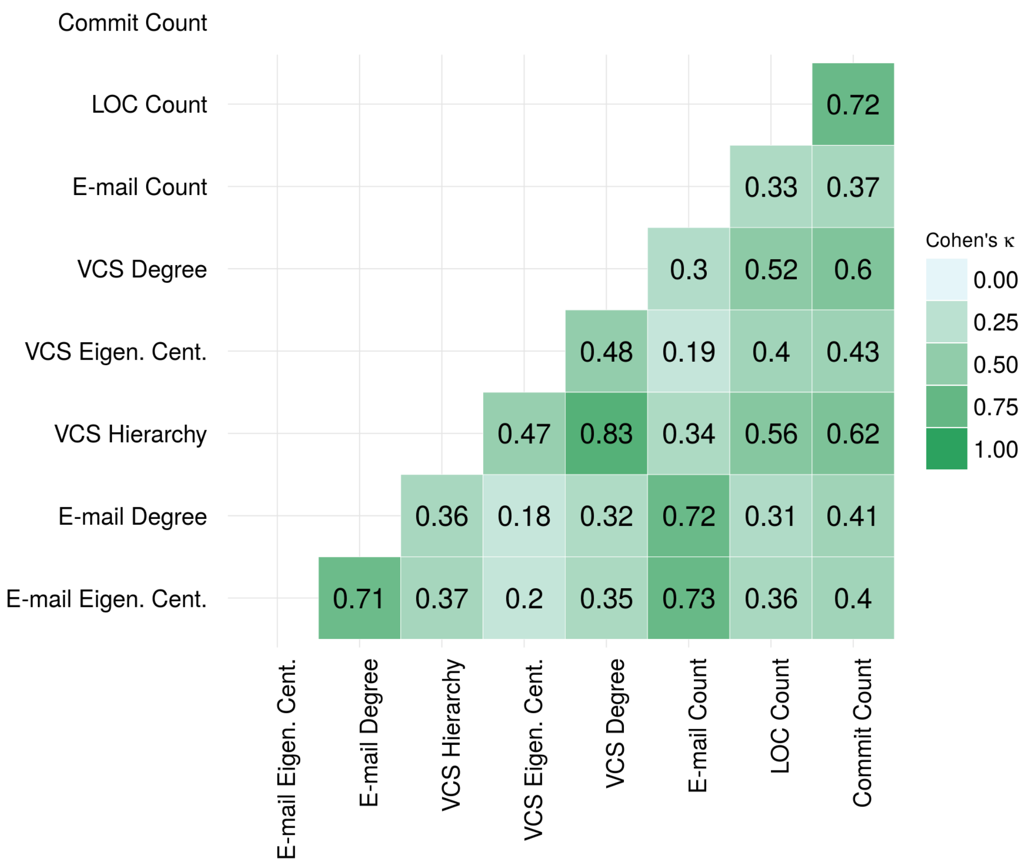


Figure 5.7: Time-averaged agreement in terms of Cohen's kappa for QEMU. The pairwise agreement is shown for the count-based and network-based operationalizations

network-based operationalizations to the ground-truth classification are shown in Table 5.3. Agreement was computed for 163 ground-truth samples provided by the survey participants. Three participant responses were eliminated because they were incomplete.

The nominal agreement values, in terms of Cohen's kappa, exceed 0 indicating that all operationalizations agree with developer perception significantly more than what is expected by chance (cf. Section 5.3.3). The highest and second highest agreement is seen in the node degree metric for the E-mail

Table 5.3: Agreement with developer perception

		Cohen's kappa	p value
Counts	Commit Count	0.387	3.12e-06
	LOC Count	0.355	1.91e-05
	E-mail Count	0.421	2.08e-05
Networks	VCS Degree	0.465	4.48e-08
	VCS Hierarchy	0.437	2.22e-07
	VCS Eigen. Cent.	0.404	1.74e-06
	E-mail Degree	0.497	8.23e-07
	E-mail Eigen. Cent.	0.427	1.26e-05

network and VCS network, respectively. The lowest agreement is seen for the count-based version-control-system metrics (Commit and LOC count). In general, all network-based operationalizations agree better (albeit in some cases only slightly) with developer perception than the basic version-control-system count-based metrics. Focusing on the comparison between different data archives, the agreement for the mailing lists metrics have even greater agreement than the corresponding version-control-system metric. However, network-based metrics always outperform the count-based metrics when the data source is fixed.

In general, the mailing list is most accurate in capturing characteristics that reflect developer perception of roles. However, in many projects communication archives are not available, and in this case a network perspective on version-control system data can closely resemble the insights (regarding developer roles) provided by the communication archive. Overall, we see that a network perspective always improves the agreement with developer perception over the simpler count-based operationalizations. To this end, *we accept H5.*

5.4.5 Further Support for a Relational Perspective

In addition to providing data for testing our hypotheses, the developer survey provides additional evidence for and insights into the usefulness of a relational perspective on developer roles. Our survey results suggest that developer roles are often defined in terms of differences in the mode of interaction between developers. For example, one developer wrote "core maintainers participate in discussions on areas outside the ones that they maintain". Only a relational perspective is able to capture this view, for example, in terms of core developers having a higher degree than peripheral developers, because they interact with developers working in areas that are distinct from the ones that they maintain. In the same vein, core developers are likely to occupy upper positions in a hierarchy, as they provide coordination bridges between the peripheral developers that have a comparatively narrow focus. Another core developer mentioned, “I may not be contributing as much as I did in past years, but I am still active and available to answer questions from and provide guidance to other developers.” Again, the developer has emphasized their role based on a mode of interaction with other developers. Another survey participant commented: “The Wine project has lots of committers and a very loose structure. It’s very hard to know who does what.” A relational view on the global organizational structure has practical value to support this kind of developer awareness that is currently missing. Beside static network properties, we argue that the temporal dimension is needed to accurately operationalize developers roles, which is also supported by survey responses: “The boundaries are fuzzy and can change over time — sometimes I’m a core developer on libvirt, while at the present I’m only a peripheral developer” or “I tend to classify contributors as regular opposed to occasional.” This is especially important as count-based operationalizations do not capture temporal relationships.

5.5 Threats to Validity

Construct Validity Quantifying the extent to which the operationalizations of developer roles represent the real world is one of the primary contributions of this work. We used the concept of mutual agreement as a testament to the validity of the operationalizations, however, one explanation for observing mutual agreement could be that all the operationalizations consistently reach the same wrong conclusion. While this would be a rather improbable explanation, we carried out a developer survey to provide additional evidence for that the operationalizations are valid.

For the network-based operationalizations, we used developer networks and network-analysis techniques to establish a relational basis for studying core and peripheral developers. This poses the threat that the networks and metrics do not accurately capture reality. This threat is minor as there is already evidence indicating that both the networks and the metrics are authentic in reflecting developer perception [JMA^{+}15; MW11]. One concern we have is regarding the unification of developers contributions, across multiple archives (i.e., mailing list and version-control system), to a single alias. However, core developers have an interest in being recognized for each contribution they make, therefore, maintaining multiple aliases would not be productive. For this reason, we think this issue has limited influence on developer classifications.

Internal Validity We quantify the agreement between different operationalizations in terms of Cohen's kappa. For these experimental conditions, we required a probabilistic definition of agreement, because a non-error-tolerant agreement metric would be too strict to yield practical results. Cohen's kappa requires some degree of interpretation though, so we have conservatively chosen thresholds that have been established in the literature.

The results of the developer survey depend partially on individual perceptions. To limit this threat, we designed the questionnaire such that multiple developers classified the same developer and we then took the average classification to limit individual bias.

External Validity The results of our study are based on the analysis of 10 open-source projects. Although, the projects do represent a broad spectrum in several dimensions, they are still limited to relatively successful, mature, and large projects. Nevertheless, the results may not be relevant to immature or very small projects. Likewise, some projects, while having significant commercial involvement (e.g., Linux), are still in the end open source and it is not yet clear if these results hold for commercial projects.

5.6 Related Work

Core–Peripheral Roles A substantial body of research on core and peripheral developers has established an understanding of the characteristics possessed by each group. Researchers have examined the core and peripheral developer roles from two distinct perspectives: from a social perspective, by studying communication and collaboration patterns [CHC08; DB05; JSW11; MVV14; MFH02], and from a technical perspective, by studying patterns of contributions of developers to technical artifacts [CWLH06; DB05; JS07; JSW11; MFH02; TRC10]. Regarding social characteristics, core developers play a central role in the communication and leadership structure [CHC08] and have substantial communication ties to other core developers, especially in projects with a small developer community (10–15 people) [MVV14; MFH02]. Regarding technical characteristics, core developers typically exhibit strong ownership over particular files that they manage, they often have detailed knowledge of the system architecture, and they have demonstrated themselves to be extremely competent [CWLH06; JS07; MFH02; TRC10]. In contrast, peripheral developers are primarily involved in identifying code-quality issues and in proposing fixes, while also participating moderately in development-related discussions [MFH02]. Since the roles of developers are not static, prior research has also investigated temporal characteristics of core and peripheral developers in terms of the advancement process to achieving core-developer status. Advancement is typically merit-based and often involves long-term, consistent, and intensive involvement in a project [CH05; JS07; JSW11; MFH02; YK03].

Operationalizing Roles Many of the aforementioned studies applied empirical methods based on interviews, questionnaires, personal experience reports, and manual inspections of data archives to identify characteristics of core and peripheral developers. An alternative line of research has attempted to operationalize the roles of core and peripheral developers using data available in software repositories, such as version-control systems [dSFD05; MFH02; OSdO$^+$12; RGH09; RG06; TRC10], bug trackers [CWLH06], and mailing lists [BGD$^+$07; OSdO$^+$12]. By operationalizing the notion of core and peripheral developers, these studies took important steps towards gaining insight that is not attainable with (more) manual approaches, including evaluating and basing conclusions on results from hundreds of projects [CH05]. Numerous studies have made use of core–peripheral operationalizations [CWLH06; JSW11; MFH02; RGH09; RG06; TRC10]. Most operationalizations are single-dimension values that represent the developer's activity level (e.g., the number of commits made, lines of code contributed, E-mail sent). A threshold is defined based on a prescribed percentile to divide the developer into either the core or peripheral group. The most commonly used approach is to count the number of commits made by each developer, and then to compute a threshold at the 80% percentile [CWLH06; MFH02; RGH09; RG06; TRC10]. This threshold was rationalized by observing that the number of commits made by developers typically follows a Zipf distribution (which implies that the top 20% of contributors are responsible for 80% of the contributions) [CWLH06]. Following a similar direction, Mockus et al. found empirical evidence for Mozilla browser and Apache Web server that a small number of developers are responsible for approximately 80% of the code modifications [MFH02]. However, in a replication study performed on FreeBSD, the results indicated that a set of "top developers" are responsible for approximately 80% of the changes, but this group does not coincide well with the elected core developer group [DB05]. More recently, attempts have been made to investigate the difference between core and peripheral developers by using basic social-network centrality metrics and a corresponding threshold [BGD$^+$07; dSFD05; OSdO$^+$12]. In these cases, developer networks have been constructed on a dyadic domain of either mutual contributions to mailing-list threads or source-code files.

Validity of Role Operationalizations While many approaches exist to classify developers into core and peripheral, no substantial evidence has been accumulated to evaluate the validity and consistency of these different operationalizations. Crowston et al. [CWLH06] investigated three operationalizations of core and peripheral developers, but they focused only on bug-tracker data and neglected code authorship entirely. Olivia et al. [OSdO$^+$12] dedicated attention on developing a more detailed characterization of so-called "key developers", which is similar to the core-developer dichotomy. They investigated mailing lists and version-control systems with three operationalizations to classify developers as core or peripheral. Their results indicate that there is some evidence of agreement between the different operationalizations, but this was only shown for a one release of a single small project with only 16 developers, in total, and 4 core developers. We improve over the state of the art by considering a larger and more diverse set of projects with larger developer communities, by using more metrics, and by analyzing, at least, one year of development, to evaluate the temporal stability of our results. Additionally, we base our operationalizations on developer network models that we have shown to exhibit real-world validity (cf. Chapter 3).

5.7 Summary

Software developers can play different roles in software projects. Information on these roles is crucial to understanding the collaborative dynamics of software projects. In particular, knowing the role of a developer provides insight regarding from whom do they likely need support or to whom could they offer support, given their current skill set and knowledge. In large, globally-distributed projects, this kind of insight can provide enormous benefits by reducing the overhead associated with developer coordination [CH13; dSR11].

In this chapter, we conducted an empirical study of 10 substantial open-source projects and established evidence that commonly used count-based operationalizations of developer roles reach consistent conclusions. In particular, we found that the pairwise agreement between the operationalizations is significant and especially high when comparing operationalizations based on the same archive type. Furthermore, the agreement is temporally stable over time, which is a further testament to its validity.

Nevertheless, while offering some utility for identifying developer roles, the insights that count-based operationalizations can provide are clearly limited, in particular, with regard to the manifold relationships between developers, which may even vary over time. As a novel contribution, we use developer networks, constructed from versions-control system (cf. Chapters 3 & 4) and mailing list data, to establish a relational perspective on developer roles in terms of the network's structural and evolutionary features.

A key hypothesis, that we confirm in this chapter, is that developer roles manifest distinctly in the organizational structure of software projects, which is substantiated by a survey among 166 developers. To this end, we have proposed a number of corresponding network metrics, such as positional stability, hierarchy, and a core–peripheral block model, to explore structural and evolutionary characteristics that emphasize differences between core and peripheral developers. Analyzing our 10 subject projects, we found that the network-based operationalizations exhibit moderate to substantial agreement with the count-based operationalizations.

While both the count-based and network-based operationalizations of developer roles hold face validity, it has not yet been shown to what extent they reflect developer perception. Based on a survey among 166 developers, we established a ground-truth classification to address this open question. We found that all operationalizations agree with developer perception, but some align more closely than others. In particular, we found that, for count-based operationalizations, mailing-list data are more accurate in representing developer perception of roles than the version-control system. Regarding network-based operationalizations, we found that using a network perspective always outperforms the corresponding count-based operationalization with respect to agreeing with developer perception.

Furthermore, our study of the temporal dimension revealed a distinction between core and peripheral developers, which is again consistent with real-world interpretations. We find this to be an important result because the count-based operationalizations do not capture temporal relationships. For example, a developer making 100 commits in one week, will appear to be equal to a developer making 2 commits per week for 50 weeks in a row, provided that the analysis window is sufficiently large.

Our results suggest that a network perspective can offer valuable insights regarding developer roles that are concealed by non-relational operationalizations. For example, the core group is comprised of the most heavily coordinated developers, and peripheral developers are more likely to coordinate with core developers than with other peripheral developers. We also found that core developers are relatively stable in the organizational structure, whereas peripheral developers tend to be more volatile. These insights are a testament to the richness of a network perspective and demonstrate a unique quality of developer networks to capture real-world phenomena with greater precision than alternative representations.

CHAPTER 6

Conclusion and Future Work

The relatively recent shift from small co-located software projects to large-scale globally distributed ones has resulted in the appearance of new factors that pose significant threats to project success. In this new globally-distributed development environment, coordination among developers constitutes one of the biggest challenges that a project faces. More specifically, the management of task interdependencies among developers is critical to developer productivity and product quality. By understanding the structure of coordination requirements among developers and how they evolve over time, we are able to conceive of strategies for improving coordination based on principled approaches rather than arbitrarily drawing samples from a collection tools and techniques. To increase the maturity of our scientific discipline, we should strive to establish a knowledge base concerning the fundamental properties of developer coordination so that we are able to propose and test theories of coordination and prescribe specific solutions to address specific coordination challenges.

For a software project to achieve a benefit from a large number of developers, software developers must be capable of working both efficiently and concurrently on development tasks. Principles of modularity and information hiding are imperative for achieving this goal, but cannot entirely eliminate the need for developers to coordinate. Regardless of how ingeniously a system is decomposed into loosely coupled parts, uncertainty present at the design phase and assumptions embedded in the implementation inevitably cause a need for developers to coordinate their efforts. By adopting a socio-technical perspective on software project analyses, one is able to gain insight into how the influence of technical decisions propagate into the project's organizational structure by altering the need for developers to coordinate. For practitioners, this kind of information can be used to estimate the organizational impact of an architectural change or whether a given project is an appropriate candidate for offshoring. For researchers, this kind of information provides insight into how and why certain methods and tools for supporting coordination succeed or fail in a given project context. For example, does introducing an instant messaging service for developers help them to become more aware of coordination requirements that are relevant to them and overcome geographical barriers that are intrinsic to globally distributed projects, or does the messaging service simply add to coordination overhead by encouraging frivolous contact with colleagues?

While a socio-technical perspective on software projects certainly has value, obtaining an accurate representation of this perspective is non-trivial. One of the major challenges stems from the intrinsic human factor and the need to model the multifaceted concept of inter-developer relationships based solely on seemingly rudimentary indicators. Software repositories provide insight into the activities of developers, but this is not the primary intended purpose of these archives. The consequence is that software repositories alone do not provide adequate information for this purpose. A version-control system provides information on what elements of the software a developer changes and at what point in time a change was made, while a mailing list provides information about which developers communicate with each other. By examining the patterns in contributions of developers to software repositories, we can get a first-order approximation to whether there exists a requirement for two developers to coordinate. By composing all pairwise interactions among every developer in a project into a single representation, we can generate a network that represents the overall coordination structure of a software project. To improve over the current state-of-the-art, we have augmented the basic information provided by software repositories to elicit insights that capture higher-order organizational properties that are an accurate depiction of real-world relationships among developers. We achieve this result by means of static-program analysis, relational abstraction, network analysis, and statistical techniques. By doing so, we have developed strategies to overcome some limitations imposed by data that are immediately available from software repositories. Using our analysis strategies, we have conducted multiple empirical studies that have led to the discovery of fundamental structural and evolutionary properties of developer coordination that shed light on previously unknown territory. With this knowledge, we now have a better understanding of the socio-technical environment that developers work within, how that environment changes overtime, and what a developer's embedding within that socio-technical environment indicates about the particular role they fulfill in the project.

6.1 Contributions

In this thesis we made the following contributions:

1. In Chapter 3 (*Community Detection and Validation with Fine-grained Developer Networks*), we investigated approaches for abstracting the activities of developers, as events in version-control systems, into a network representation. Specifically, we focused attention to understanding the influence of various heuristics, which are used to identify whether two developers are engaged in interdependent activities, on the real-world authenticity of the developer network edges and high-order structures (i.e. developer communities). Our investigation included heuristics based on: localizing changes at a file-level granularity, manually reported references to developers' participation in a change, and commit metadata regarding committer and author participation a change. Additionally, we proposed a fine-grained heuristic based on localizing changes at a source-code entity granularity. To study the developer networks' higher-order structure, we applied sophisticated community-detection algorithms that rely on principles of unsupervised machine learning and are capable of operating on weighted and directed graphs. Since the output of community detection algorithms are not definitive proof that developers are arranged according to a non-random organizational principle, we devised a test to evaluate the statistical significance of the community structure. Finally, we performed a survey of 53 open-source developers to determine whether the networks are an accurate depiction of developer perception.

   We evaluated our fine-grained approach against the file-level heuristic on 10 diverse open-source projects, with complex and active histories, from a variety of domains, written in various programming languages, and of different sizes. We found that developer networks constructed using our fine-grained approach exhibited statistically significant community structure. In contrast, developer networks constructed using the file-level heuristic exhibited a greater density of edges among groups of developers, thereby concealing the statistically significant community structure. In essence, the finer-grained heuristic enabled the identification of an organizational principle that was previously unobserved in open-source projects.

   From our survey of 53 open-source developers, we learned that most developers agree that the fine-grained network accurately depicts reality and the developer communities have real-world meaning. Furthermore, we found that the predominant source of error in our approach was from missing links; the links that were identified are largely accurate. We were able to show that, while the finer-granularity of our approach inherently sacrifices some edges, only a small percentage of edges concealing the community structure in the file-based networks are authentic. Given the abstract nature of a human-centric concept, such as coordination and community structure, and our fully automated method of detection, we find these results to be supportive of the validity of our approach.

2. Equipped with an authentic representation of developer coordination, we continued in Chapter 4 (*Evolutionary Trends of Developer Coordination*) by adding in the time component to study developer networks' dynamic properties. To generate a temporally ordered sequence of developer networks, we paired our fine-grained approach with a sliding-window technique so that we are able to capture development continuity among subsequent networks. We then applied our approach to study 18 substantial open-source projects. Our first research question focused on the observable changes in three fundamental organizational principles, namely, scale freeness, modularity, and hierarchy. Our second research question focused on the relationship between the developer networks structural properties and project scale. The results of the study suggest that the organizational structure of large projects is constrained to evolve towards a state that balances the costs and benefits of developer coordination, and the mechanisms used to achieve this state depend on the project's scale.

   Based on our longitudinal study of 18 open-source software projects, we found that, in projects exceeding 50 developers, the coordination structure becomes scale free. Interestingly, all projects that were capable of achieving long-term sustained growth, in terms of the number contributing of developers, were also scale-free developer networks. Scale freeness is potentially one possible arrangement of developer coordination requirements that leads to sustainable project growth or is possibly an indication of a healthy socio-technical environment. We also found that there is a tendency for an increasing number of coordination requirements to appear among groups of developers, but the increasing trend is likely limited by a particular upper bound, where coordination requirements exist between roughly half of every developers neighbors. The implication is that as a project matures, the coordination burden on developers increases. On this basis, we argued that the amount of necessary for coordinating with other developer and the techniques used for achieving a coordinated effort are presumably different depending on the project's state of maturity. Additionally, we discovered that developers are hierarchically arranged in the early phases of a project, but in later phases the global hierarchy vanishes and a hybrid structure emerges, where core developers form the hierarchy and peripheral developers exist outside the hierarchy. It is plausible that global hierarchies are simply not flexible enough during later project phases because of the introduction of many peripheral developers that are highly volatile in nature. With this result, we demonstrated that core and peripheral developers—which are traditionally defined based on their level of participation—also differ in how they are structurally embedded in the project's coordination structure. Overall, the adaptations that we observed in the structural features balance the opposing constraints of supporting effective coordination and achieving robustness to developer withdrawal. Finally, we discussed how these structural features enable a project in benefiting from a large, but volatile, peripheral developer group, while at the same time, supporting effective coordination and regularity between the much more stable core developer group. From these results, it is clear that significant structural changes occur in the coordination structure of a project over time, and particularly as developers are added. These insights provide valuable information to software engineering practitioners by highlighting the impact that adding developers has on the coordination structure. With this knowledge we can begin to establish strategies for integrating new developers that try to minimize the disruption to the existing coordination structure.

Apart from the general patterns that explain the majority of subject projects, we also noted a number of interesting deviations from the general patterns. For example, during a period of time, when Firefox was experiencing notable project delays and turmoil within the developer community, we observed that scale freeness suddenly disappeared. In Node.Js, there was an oscillatory behavior to the number of contributing developers and the scale-freeness property was lost whenever the project was not in a growing state. Finally, for the few projects that never became scale free, or only for a brief time, a developer group larger than 60 was never sustainable and the number of contributing developers decreased shortly after reaching a maximum, as was the case for PHP, jQuery, and Apache HTTP. It was often the case that projects deviating significantly from the general patterns were experiencing other negative project conditions such as significant loss in the number of active developers. Further exploration of structural and evolutionary differences between the developer networks of successful and unsuccessful projects is a promising avenue for future work.

3. Based on our validated approach for constructing developer networks and the structural and temporal patterns that govern the network's evolution,

in Chapter 5 (*Classifying Developers into Core and Peripheral Roles*) we developed and evaluated approaches for classifying developers into the role they fulfill in a project. In an empirical study of 10 substantial open-source projects, we established evidence that operationalizations based on a network representation are capable of outperforming the more commonly applied standard variable representations.

As a first contribution, we determined that the prevalent count-based operationalizations of developer roles generate consistent results. In particular, we found that the pairwise agreement between the operationalizations is statistically significant and especially high when comparing two operationalizations based on the same archive type. Furthermore, the agreement is temporally stable over time, which is a further testament to its validity.

Nevertheless, while offering some utility for identifying developer roles, the insights count-based operationalizations can provide are clearly limited, in particular, with regard to the manifold relationships between developers, which may even vary over time. As a novel contribution, we use developer networks to establish a relational perspective on developer roles. A key hypothesis is that developer roles should manifest distinctly in the organizational structure, which is also substantiated by a survey we conducted among 166 open-source developers and our observations of an organizational dichotomy in the network evolution study (cf. Chapter 4). To this end, we have proposed a number of corresponding network metrics, such as positional stability, hierarchy, and a core–peripheral block model, to explore structural characteristics that capture differences between core and peripheral developers.

While both the count-based and network-based operationalizations of developer roles hold face validity, it has not yet been shown to what extent they reflect developer perception. Based on a survey among 166 developers, we established a ground-truth classification of developer roles to address this open question. We found that the operationalizations exhibit moderate to substantial agreement with developer perception, in terms of Cohen's kappa. More specifically, we found that, for count-based operationalizations, mailing-list data are more accurate in representing developer perception of roles than the version-control system data. Regarding network-based operationalizations, we found that adopting a network perspective, instead of the alternative, always increases the agreement with developer perception.

Furthermore, our study of the temporal dimension revealed a distinction between core and peripheral developers, which is again consistent with real-world interpretations. We find this to be an important result because the count-based operationalizations do not capture temporal relationships. For example, a developer making 100 commits in one week, will appear to be equal to a developer making 2 commits per week for 50 weeks in a row, provided that the analysis window is sufficiently large.

Our results suggest that a network perspective can offer valuable insights regarding developer roles that are concealed by non-relational operationalizations. For example, the core group is comprised of the most heavily coordinated developers, and peripheral developers are more likely to coordinate with core developers than with other peripheral developers. We also found that core developers are relatively stable in the organizational structure, whereas peripheral developers tend to be more volatile.

With the realization of an approach for constructing authentic developer networks, we have made a contribution to establishing the ground-work needed to study the fundamental properties of developer coordination. We have demonstrated that our developer networks are structured according to a number of organizational principles and identified common evolutionary trends of successful open-source projects. Lastly, we have shown that developer networks contain rich inter-developer insights that are concealed by non-relational perspectives and that network representations of developer activity are able to outperform standard variable representations in the task of classifying developers according to the role they fulfill.

6.2 Future Work

In the process of conducting the research found in this dissertation, we have identified four promising future research directions:

- Alternative Views on Coordination
- Coordination Over Complete Software Life Cycle
- Analysis of Developer Network Structure Using Manual Inspections & Interviews
- A Developer Network Growth Model

6.2.1 Alternative Views on Coordination

One of the major contributions in this dissertation is an approach for constructing developer networks that are an authentic representation of the real-world. Our approach generates a single view, but there are likely alternative views that hold equally valuable information concerning socio-technical aspects of software development. The basic elements required to construct a developer network are: (1) a traceable link between an artifact and the developers that contribute to the artifact and (2) a notion of artifact relationships. By drawing a selection from the wealth of different artifacts and artifact relationships, one is presumably capable of generating alternative views on developer coordination. For example, a software feature—a collection of code that implements a particular requirement—raises the artifact concept to a semantic level. It is plausible that features correspond to artifacts that imply a substantial coordination need among developers contributing to the feature, however, this is not yet known. The semantic quality of features is particularly interesting because it could better represent developers' perception of related code than files or functions. Fortunately tools such as cppstats[1] exist to identify features implemented with #ifdef directives, substantially lowering the barrier to conducting feature-based analyses [LvRK^{+}13]. Alternatively, using structural and evolutionary dependencies among files and clustering algorithms, high-order relationships between sets of files, termed *design rule spaces*, can be identified [XCK14]. These so-called design rules spaces are a reflection of the architectural decisions that decouple a system into modules. Design rules spaces are a sensible candidate artifact to explore the coordination requirements among developers that arise from architectural decisions. In this approach, the design rules spaces provide the information regarding files and file relations, and the version-control system provides the link between developers and files they contribute to. There are many unexplored avenues in terms of how different coupling mechanisms influences the need for developers to coordinate as well. The only exception is co-change coupling—a coupling mechanism based on files that are frequently involved in a common commit—which has already been shown to impact developer productivity and code quality [CH13]. However, the influence of dynamic dependencies, semantic dependencies, and remote procedure dependencies, among others [CSA11; MMP00], on developer coordination is virtually unexplored. Other opportunities for future work are exploring dependencies among artifacts that are less often recognized, but nonetheless have shown to influence the need for developers to coordinate. The primary sources of these other dependencies

[1]`http://fosd.de/cppstats/`

stemmed from scheduling strategies, management of shared resources, and state synchronization [BMB+07].

6.2.2 Coordination Over Complete Software Life Cycle

The approaches, analyses, and insights we presented in this dissertation are primarily focused on the coding phases of the software life cycle without particular attention to which changes are new feature code, maintenance, or testing related. While these phases are typically regarded as the longest and most expensive [BR00], there are certainly rational arguments for applying our approaches on other phases. It is plausible that improper assignment of tasks among people, and consequently, a lack of adequate coordination in phases prior to coding, manifests as software quality problems in later phases. Since the cost of fixing defects becomes exorbitantly more expensive in later phases compared to earlier ones [Mad94], there is a substantial interest in addressing the source of a problem rather than its consequence. An opportunity for future work is to examine the coordination requirements implied by the artifacts generated during other phases of the software life cycle (e.g., requirements documents, use cases, data models, workflow diagrams, organizational charts, etc.). For example, with information about which individuals are responsible for defining which software requirements and a notion of requirement interdependence, a network representing coordination among individuals in the requirements engineering phase could be generated. Essentially, the same idea could be applied to any phase provided that artifacts are generated by individuals, traceability exists between people and the artifacts, and notions of interdependencies among the artifacts that imply a coordination requirement can be defined.

Another promising avenue is to apply our approaches to the transitions between subsequent software life-cycle phases by focusing on the hand offs that occur around artifacts. For example, the individuals responsible for requirements analysis can be different from those doing the design work. In the transition between theses phases, ideas, concepts, and decisions need to persist across those different groups of people. The artifacts (e.g., requirements documents) may contain ambiguities or are in someway insufficient to completely convey all the required information for the design work to proceed with perfect adherence to the intentions of the requirements engineers. When the designers fail to understand the intentions of the requirements engineers, the design likely will not fully address the requirement, but this may only be discovered after the implementation phase. By representing the hand offs that occur between the individuals participating in different phases of the software life cycle as a coordination network, we may better understand where there are likely to be coordination needs among people. Information contained in the coordination network spanning the hand offs between requirements and design phases could help to achieve a smoother transition between groups involved in different phases by mitigating coordination breakdowns. In the Mars Climate Orbiter disaster, the primary cause of failure was that two different suppliers, that were geographical separated by a substantial distance, producing two distinct but coupled components failed to establish a common understanding of the requirements regarding units of measurement [Mar]. A coordination need would have been recognized by virtue of both groups needing to adhere to a common requirement. The main point is that mistakes will occur, but communication channels should exists between appropriate individuals to catch and correct the mistakes when they do inevitably occur. The coordination networks serve as means to identify who are those appropriate individuals so that not every single person in the project needs to communicate and overwhelm the process frivolous interactions.

6.2.3 Analysis of Developer Network Structure Using Manual Inspections & Interviews

As part of our approach to studying developer networks, we conducted developer surveys to validate and gain insight into the developer network's structural properties. In Chapter 3, we made use of surveys to establish real-world validity of the network edges and developer communities. Similarly, in Chapter 5 we used developer surveys to test the validity of results produced by our network-based operationalizations of developer roles. It is important to remain cognizant of the fact that developer networks are an abstraction of complex real-world phenomena. To a large extent, the value offered by developer networks is dependent on their ability to accurately capture nuances about the real-world relationships among developers. For this reason, it is imperative that more work is done on the feedback loop leading from the network insights back to the real-world phenomena we originally intended to abstract. In particular, we identified a number of organizational principles in Chapter 4 such as scale freeness, modularity, and hierarchy. In each of these properties is an opportunity to perform an in-depth study to better understand the effect of the property on a project. For example, there are theoretic robustness benefits to a scale-free network, but it is not yet known if such benefits are realized in a software project. In terms of hierarchy, the lack of a hierarchical arrangement in peripheral developers should allow the organizational structure to be more flexible. Again, the flexible quality has not been shown to exist. In terms of modularity, we observed a general increasing trend. It is not clear if this coincides with increase familiarity among developers that are densely interconnected, or increase communication on the mail list. These are a couple examples of many important questions that could be answered by performing an in-depth analysis through a combination manual inspections of the developer networks and corresponding interviews with the developers to better understand real-world implications of the structural properties we have identified.

6.2.4 A Developer Network Growth Model

In Chapter 4, we discussed relationships between the structural properties of developers networks and the network's growth over time. We noted several intriguing results, one of which was that the nodes clustering coefficients typically increase as the network size increases. We also found that networks exhibiting this feature are also simultaneously scale free. This particular result is interesting because the predominant growth model used to explain the formation of scale free networks is preferential attachment (cf. Section 2.3.4), however, the model of preferential attachment does not explain the increasing clustering with increasing network size. From this result, we reason that the growth of developer networks is presumably described by an alternative growth model. A direction for future work is to identify a growth model that better describes the observations we have found. In essence, the elements of a growth model provide a description of how edges and nodes are added and removed in the network over time. Typically, the growth process is described probabilistically. Probability distribution functions are used to define the mechanism by which nodes and edges are added or removed based on local or global network properties. A major component of developing the growth model is to find what network properties should be used in the parametrization of the growth model. A basic example of such a parameterization is to use the local property of node degree to define the probability that two nodes form an edge. The developer network evolution study we performed in Chapter 4 provides valuable information as to what an appropriate parameterization may be, but many open questions remain. The evolution study provides observations on the developer network's growth over sequential time periods. This information could be used to either infer a growth model from the observed data or to test the fitness of a hypothesized model against real-world data [PDS$^+$12].

In the fields of ecology and epidemiology, knowledge about growth models concerning networks have proven to offer both valuable and practical insights. In ecology, network growth models provide insight into which specific conditions (e.g., in terms of birth rates or competition for resources) lead to the decimation of species [May72]. In epidemiology, network growth models have enabled disease prevention techniques through modeling the spread of infectious diseases with networks, which has been instrumental in establishing effective public health policy and protocols for mitigating the transmission of infectious diseases [KE05; May72]. In the case of developer networks, a growth model can help to reason about the complex microinteractions among developers at a higher level of abstraction. The microinteractions in developer networks encompass the small-scale interactions among pairs of developers. It is likely the case that the diverse interactions among pairs of developers gives rise to a common, and relatively simple, collective behavior in much the same way that the diverse interactions among a large number of independent random variables are ubiquitously captured by the Gaussian distribution, a phenomenon that is well known and proven by the central limit theorem [DN15]. If this kind of universality also exists in developer networks, the growth model would allow us to forgo a deep understanding underlying mechanisms driving the system at a microscopic level, yet still be capable of understanding the overall collective behavior to a high degree of accuracy. Since the developers that make up a network are not independent in their behavior—we have found first had evidence of this in many cases where correlations among sets of developers often give rise to higher-order structure—developer networks are likely governed by different models than those that are justified using the central limit theorem. One distribution that has seen success in modeling complex systems with correlated random variables, and presumably has applications in modeling network growth, is the Tracy Widom distribution [TW02]. In the field of ecology, the Tracy Widom distribution has played a key role in determining the conditions under which a set of species occupying a local region is in a stable and unstable phase and where the phase transition occurs.

The practical consequences of identifying a growth model for developer coordination networks could very well result in a valuable contribution to the field of software engineering. If an appropriate growth model is found, it could provide insight into important phases and phase transitions that occur in the coordination structure during software development. That in turn could provide hints at healthy and unhealthy growth processes. In the domain of ecology, notions of phase transitions have been used to show that certain conditions lead a phase of equilibrium among different species, while other conditions lead another phase where the species' populations have a tendency to rapidly approach zero or infinity. Interestingly, the transition is not at all gradual. Instead, there appears to be a tipping point that signifies an abrupt change between these two drastically different phases [May72]. It may be the case that some phases in developer networks correspond to a similar type of instability that results in software quality problems stemming from coordination breakdowns. With a deeper understanding of these phases, one may be able to even control the network dynamics towards a more positive direction by encouraging phase transitions towards a healthier or more stable phase. Based on Conway's law, there is reason to believe that changes made to the software architecture may even trigger destabilization in the developer coordination structure [Con68]. At the very least, awareness of when a project is in an unstable phase is already a benefit. During unstable phases, one could dedicate more effort and time towards coordination activities to avoid coordination breakdowns during this fragile state.

Bibliography

[AB02] R. Albert and A.-L. Barabási, “Statistical mechanics of complex networks”, *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[AKC01] E. B. Allen, T. M. Khoshgoftaar, and Y. Chen, “Measuring coupling and cohesion of software modules: An information-theory approach”, in *Proceedings of the International Symposium on Software Metrics (METRICS)*, IEEE Computer Society, 2001.

[AGMZ11] H. Almeida, D. Guedes, W. Meira, and M. J. Zaki, “Is there a best quality metric for graph clusters?”, in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2011, pp. 44–59.

[And11] M. Andreessen, “Why software is eating the world”, *The Wall Street Journal*, Aug. 20, 2011.

[Atk70] A. B. Atkinson, “On the measurement of inequality”, *Journal of Economic Theory*, vol. 2, no. 3, pp. 244–263, 1970.

[BHKL06] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, “Group formation in large social networks: Membership, growth, and evolution”, in *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*, ACM, 2006, pp. 44–54.

[BR$^+$99] R. Baeza-Yates, B. Ribeiro-Neto, *et al.*, *Modern information retrieval.* Addison-Wesley, 1999.

[BEN$^+$93] U. Banerjee, R. Eigenmann, A. Nicolau, D. A. Padua, *et al.*, “Automatic program parallelization”, *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, 1993.

[BA99] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks", *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[BMB+07] M. Bass, V. Mikulovic, L. Bass, J. Herbsleb, and M. Cataldo, "Architectural misalignment: An experience report", in *Proceedings of the IEEE/IFIP Conference on Software Architecture*, IEEE Computer Society, 2007.

[BDO+13] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 692–701.

[BR00] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: A roadmap", in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2000, pp. 73–87.

[BGD+06] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks", in *Proceedings of the International Workshop on Mining Software Repositories*, ACM, 2006, pp. 137–143.

[BGD+07] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, "Open borders? Immigration in open source projects", in *Proceedings of the International Workshop on Mining Software Repositories*, IEEE, 2007.

[BNM+11] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality", in *Proceedings of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM, 2011, pp. 4–14.

[BPD+08] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects", in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, ACM, 2008, pp. 24–35.

[BRB+09] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git", in *Proceedings of the IEEE International Working Conference on Mining Software Repositories*, IEEE Computer Society, 2009, pp. 1–10.

[Bis06] C. M. Bishop, *Pattern recognition and machine learning.* Springer, 2006.

[BLM$^+$06] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang, “Complex networks: Structure and dynamics”, *Physics reports*, vol. 424, no. 4, pp. 175–308, 2006.

[Boe89] B. W. Boehm, Ed., *Software risk management.* IEEE, 1989.

[Bon87] P. Bonacich, “Power and centrality: A family of measures”, *American Journal of Sociology*, vol. 92, no. 5, pp. 1170–1182, 1987.

[BE00] S. P. Borgatti and M. G. Everett, “Models of core/periphery structures”, *Social networks*, vol. 21, no. 4, pp. 375–395, 2000.

[BE05] U. Brandes and T. Erlebach, *Network analysis: Methodological foundations.* Springer, 2005.

[BGW03] U. Brandes, M. Gaertler, and D. Wagner, “Experiments on graph clustering algorithms”, in *Proceedings of the European Symposium on Algorithms*, Springer, 2003, pp. 568–579.

[Bro78] F. P. Brooks Jr., *The mythical man-month: Essays on software engineering.* Addison-Wesley, 1978.

[CSA11] T. B. Callo Arias, P. Spek, and P. Avgeriou, “A practice-driven systematic review of dependency analysis solutions”, *Empirical Software Engineering*, vol. 16, no. 5, pp. 544–586, 2011.

[CH13] M. Cataldo and J. D. Herbsleb, “Coordination breakdowns and their impact on development productivity and software failures”, *IEEE Transactions on Software Engineering*, vol. 39, no. 3, pp. 343–360, 2013.

[CHC08] M. Cataldo, J. D. Herbsleb, and K. M. Carley, “Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity”, in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, ACM, 2008, pp. 2–11.

[CMRH09] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, “Software dependencies, work dependencies, and their impact on failures”, *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864–878, 2009.

[CSN09] A. Clauset, C. R. Shalizi, and M. E. J. Newman, “Power-law distributions in empirical data”, *SIAM Review*, vol. 51, no. 4, pp. 661–703, 2009.

[CEbH01] R. Cohen, K. Erez, D. ben-Avraham, and S. Havlin, “Breakdown of the internet under intentional attack”, *Physical Review Letters*, vol. 86, pp. 3682–3685, 16 Apr. 2001.

[Con68] M. E. Conway, “How do committees invent”, *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[Cor] J. Corbet, `http://www.linuxfoundation.org/content/22-lifecycle-patch`, Accessed: 2016-04-28.

[CH05] K. Crowston and J. Howison, “The social structure of free and open source software development”, *First Monday*, vol. 10, no. 2, 2005.

[CWLH06] K. Crowston, K. Wei, Q. Li, and J. Howison, “Core and periphery in free/libre and open source software team communications”, in *Proceedings of the International Conference on System Sciences*, IEEE, 2006, pp. 118–127.

[CKI88] B. Curtis, H. Krasner, and N. Iscoe, “A field study of the software design process for large systems”, *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1287, 1988.

[dAS09] B. de Alwis and J. Sillito, “Why are software projects moving from centralized to decentralized version control systems?”, in *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*, IEEE Computer Society, 2009, pp. 36–39.

[dSR11] C. R. B. de Souza and D. F. Redmiles, “The awareness network, to whom should I display my actions? and, whose actions should I monitor?”, *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 325–340, 2011.

[dSQTR07] C. R. de Souza, S. Quirk, E. Trainer, and D. F. Redmiles, “Supporting collaborative software development through the visualization of socio-technical dependencies”, in *Proceedings of the International ACM Conference on Supporting Group Work*, ACM, 2007, pp. 147–156.

[dSFD05] C. de Souza, J. Froehlich, and P. Dourish, “Seeking the source: Software source code as a social and technical artifact”, in *Proceedings of the International Conference on Supporting Group Work*, ACM, 2005, pp. 197–206.

[DOS99] C. DiBona, S. Ockman, and M. Stone, Eds., *Open sources: Voices from the open source revolution.* O’Reilly, 1999.

[DB05] T. T. Dinh-Trong and J. M. Bieman, “The FreeBSD project: A replication case study of open source development”, *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 481–494, 2005.

[DM03] S. N. Dorogovtsev and J. F. F. Mendes, *Evolution of networks: From biological nets to the internet and www*. Oxford University Press, 2003.

[DN15] R. M. D’Souza and J. Nagler, “Anomalous critical and supercritical phenomena in explosive percolation”, *Nature Physics*, vol. 11, no. 7, pp. 531–538, 2015.

[EM12] E. Eaton and R. Mansbach, “A spin-glass model for semi-supervised community detection”, in *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI Press, 2012, pp. 900–906.

[ES13] J. Eilperin and S. Somashekhar, “Private consultants warned of risks before Healthcare.gov’s Oct. 1 launch”, *The Washington Post*, Nov. 18, 2013.

[EK08] K. E. Emam and A. G. Koru, “A replicated survey of it software project failures”, *IEEE Software*, vol. 25, no. 5, pp. 84–90, 2008.

[EWSG94] S. D. Eppinger, D. E. Whitney, R. P. Smith, and D. A. Gebala, “A model-based method for organizing tasks in product development”, *Research in Engineering Design*, vol. 6, no. 1, pp. 1–13, 1994.

[ER59] P. Erdős and A. Rényi, “On random graphs”, *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

[ESKH07] J. A. Espinosa, S. A. Slaughter, R. E. Kraut, and J. D. Herbsleb, “Familiarity, complexity, and team performance in geographically distributed software development”, *Organization Science*, vol. 18, no. 4, pp. 613–630, 2007.

[FO00] N. E. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system”, *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[FLG00] G. W. Flake, S. Lawrence, and C. L. Giles, “Efficient identification of web communities”, in *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, ACM, 2000, pp. 150–160.

[FSE07] W. Fong Boh, S. A. Slaughter, and J. A. Espinosa, "Learning from experience in software development: A multilevel analysis", *Management Science*, vol. 53, no. 8, pp. 1315–1331, 2007.

[For10] S. Fortunato, "Community detection in graphs", *Physics Reports*, vol. 486, no. 3–5, pp. 75–174, 2010.

[FB07] S. Fortunato and M. Barthélemy, "Resolution limit in community detection", *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, 2007.

[FPB$^+$15] M. Foucault, M. Palyart, X. Blanc, G. C. Murphy, and J.-R. Falleri, "Impact of developer turnover on quality in open-source software", in *Proceeding of the International Symposium on Foundations of Software Engineering*, ACM, 2015.

[Fra11] M. Franceschet, "Pagerank: Standing on the shoulders of giants", *Communications of the ACM*, vol. 54, no. 6, pp. 92–101, 2011.

[GGH$^+$07] R. Garcia, P. Greenwood, G. Heineman, R. Walker, Y. Cai, H. Y. Yang, E. Baniassad, C. V. Lopes, C. Schwanninger, and J. Zhao, "Assessment of contemporary modularization techniques", *ACM SIGSOFT Software Engineering Notes*, vol. 35, pp. 31–37, 2007.

[GMZ03] C. Gkantsidis, M. Mihail, and E. Zegura, "The markov chain simulation method for generating connected power law random graphs", in *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2003.

[GT00] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study", in *Proceedings of the International Conference on Software Maintenance*, IEEE, 2000, pp. 131–140.

[Ham94] J. D. Hamilton, *Time series analysis*. Princeton University Press, 1994, vol. 2.

[HC90] R. M. Henderson and K. B. Clark, "Architectural innovation: The reconfiguration of existing product technologies and the failure of established firms", *Administrative Science Quarterly*, pp. 9–30, 1990.

[HG99] J. D. Herbsleb and R. E. Grinter, "Splitting the organization and integrating the code: Conway's law revisited", in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 1999, pp. 85–95.

[HM03a] J. D. Herbsleb and A. Mockus, “An empirical study of speed and communication in globally distributed software development”, *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 481–494, 2003.

[HM03b] J. D. Herbsleb and A. Mockus, “Formulation and preliminary test of an empirical theory of coordination in software engineering”, in *Proceedings of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Helsinki, Finland: ACM, 2003, pp. 138–137.

[HMFG00] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter, “Distance, dependencies, and delay in a global collaboration”, in *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, ACM, 2000, pp. 319–328.

[HPB05] J. D. Herbsleb, D. J. Paulish, and M. Bass, “Global software development at siemens: Experience from nine projects”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2005, pp. 524–533.

[Hey07] F. Heylighen, “Why is open access development so successful? stigmergic organization and the economics of information”, *Open Source Jahrbuch*, 2007.

[Hie] D. Hiebert, *Exuberant ctags*, `http://ctags.sourceforge.net/`, Accessed: 2016-04-25.

[HM06] P. Hinds and C. McGrath, “Structures that work: Social structure, work structure and coordination ease in geographically distributed teams”, in *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, ACM, 2006, pp. 343–352.

[HSL$^+$98] N. E. Huang, Z. Shen, S. R. Long, M. C. Wu, H. H. Shih, Q. Zheng, N.-C. Yen, C. C. Tung, and H. H. Liu, “The empirical mode decomposition and the hilbert spectrum for nonlinear and non-stationary time series analysis”, *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1971, pp. 903–995, 1998.

[HL05] S.-K. Huang and K.-m. Liu, “Mining version histories to verify the learning process of legitimate peripheral participants”, in *Proceedings of the International Workshop on Mining Software Repositories*, ACM, 2005, pp. 1–5.

[Hum96] W. Humphrey, *Introduction to the personal software process*. Addison-Wesley Professional, 1996.

[Hus95] M. A. Huselid, "The impact of human resource management practices on turnover, productivity, and corporate financial performance", *Academy of Management journal*, vol. 38, no. 3, pp. 635–672, 1995.

[HPN10] P. Hynninen, A. Piri, and T. Niinimaki, "Off-site commitment and voluntary turnover in GSD projects", in *Proceedings of the International Conference on Global Software Engineering*, IEEE, 2010, pp. 145–154.

[JS07] C. Jensen and W. Scacchi, "Role migration and advancement processes in OSSD projects: A comparative case study", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2007, pp. 364–374.

[JSW11] C. Jergensen, A. Sarma, and P. Wagstrom, "The onion patch: Migration in open source ecosystems", in *Proceedings of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM, 2011, pp. 70–80.

[JSS11] A. Jermakovics, A. Sillitti, and G. Succi, "Mining and visualizing developer networks from version control systems", in *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*, ACM, 2011, pp. 24–31.

[JAHM17] M. Joblin, S. Apel, C. Hunsen, and W. Mauerer, "Classifying developers into core and peripheral: An empirical study on count and network metrics", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2017.

[JAM17] M. Joblin, S. Apel, and W. Mauerer, "Evolutionary trends of developer coordination: A network approach", *Empirical Software Engineering*, pp. 1–45, 2017.

[JMA$^+$15] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle, "From developer networks to verified communities: A fine-grained approach", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2015, pp. 563–573.

[KCM07] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution", *Journal of Software Maintenance and Evolution*, vol. 19, no. 2, pp. 77–131, 2007.

[KVV04] R. Kannan, S. Vempala, and A. Vetta, "On clusterings: Good, bad and spectral", *Journal of the ACM*, vol. 51, no. 3, pp. 497–515, 2004.

[KE05] M. J. Keeling and K. T. Eames, "Networks and epidemic models", *Journal of the Royal Society Interface*, vol. 2, no. 4, pp. 295–307, 2005.

[Kni02] J. C. Knight, "Safety critical systems: Challenges and directions", in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2002, pp. 547–550.

[Koc04] S. Koch, "Profiling an open source project ecology and its programmers", *Electronic Markets*, vol. 14, no. 2, pp. 77–88, 2004.

[KW06] G. Kossinets and D. J. Watts, "Empirical analysis of an evolving social network", *Science*, vol. 311, no. 5757, pp. 88–90, 2006.

[Kot14] J. P. Kotter, *Accelerate: Building strategic agility for a faster-moving world.* Harvard Business Review Press, 2014.

[KS95] R. E. Kraut and L. A. Streeter, "Coordination in software development", *Communications of the ACM*, vol. 38, no. 3, pp. 69–81, 1995.

[Kum02] V. Kumar, *Introduction to parallel computing.* Addison-Wesley, 2002.

[LRRF11] A. Lancichinetti, F. Radicchi, J. J. Ramasco, and S. Fortunato, "Finding statistically significant communities in networks", *PLoS ONE*, vol. 6, no. 4, pp. 1–18, Apr. 2011.

[LK77] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data", *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.

[LRW$^+$97] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view", in *Proceedings of the International Symposium on Software Metrics (METRICS)*, IEEE, 1997.

[LR01] M. M. Lehman and J. F. Ramil, "Rules and tools for software evolution planning and management", *Annals of Software Engineering*, vol. 11, no. 1, pp. 15–44, 2001.

[LR03] M. M. Lehman and J. F. Ramil, "Software evolution: Background, theory, practice", *Information Processing Letters*, vol. 88, no. 1-2, pp. 33–44, 2003.

[LLM10] J. Leskovec, K. J. Lang, and M. Mahoney, "Empirical comparison of algorithms for network community detection", in *Proceedings of the International Conference on World Wide Web*, ACM, 2010, pp. 631–640.

[LSS05] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers: Data collection techniques for software field studies", *Empirical Software Engineering*, vol. 10, no. 3, pp. 311–341, 2005.

[LvRK$^+$13] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software", in *Proceedings of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM, 2013, pp. 81–91.

[Lik32] R. Likert, "A technique for the measurement of attitudes", *Archives of Psychology*, vol. 22, pp. 1–55, 1932.

[LRGH09] L. López-Fernández, G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, "Applying social network analysis techniques to community-driven libre software projects", *Integrated Approaches in Information Technology and Web Engineering: Advancing Organizational Knowledge Sharing*, vol. 1, pp. 28–50, 2009.

[LRG$^+$04] L. López-Fernández, G. Robles, J. M. Gonzalez-Barahona, *et al.*, "Applying social network analysis to the information in CVS repositories", in *Proceedings of the International Workshop on Mining Software Repositories*, 2004, pp. 101–105.

[LSV08] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software", *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 1, p. 2, 2008.

[Mad94] R. J. Madachy, "A software project dynamics model for process cost, schedule and risk assessment", PhD thesis, 1994.

[MC90] T. W. Malone and K. Crowston, "What is coordination theory and how can it help design cooperative work systems?", in *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, ACM, 1990, pp. 357–370.

[MVV14] C. Manteli, B. Van Den Hooff, and H. Van Vliet, "The effect of governance on global software development: An empirical research in transactive memory systems", *Information and Software Technology*, vol. 56, no. 10, pp. 1309–1321, 2014.

[Mar] Mars Climate Orbiter Mishap Investigation Board, *Mars climate orbiter mishap investigation board phase I report*, `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`, Accessed: 2016-04-25.

[MRGO08] J. Martinez-Romo, G. Robles, J. M. Gonzalez-Barahona, and M. Ortuño-Perez, "Using social network analysis techniques to study collaboration between a FLOSS community and a company", in *Proceedings of the International Conference on Open Source Systems*, Springer, 2008, pp. 171–186.

[Mau08] W. Mauerer, *Professional linux kernel architecture.* Wrox Press, 2008.

[May72] R. M. May, "Will a large complex system be stable?", *Nature*, vol. 238, pp. 413–414, 1972.

[MMP00] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of software connectors", in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2000, pp. 178–187.

[MW09] A. Meneely and L. Williams, "Secure open source collaboration: An empirical study of linus' law", in *Proceedings of the ACM Conference on Computer and Communications Security*, ACM, 2009, pp. 453–462.

[MW11] A. Meneely and L. Williams, "Socio-technical developer networks: Should we trust our measurements?", in *Proceedings of the International Conference on Software Engineering*, ACM, 2011, pp. 281–290.

[MWSO08] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis", in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, ACM, 2008, pp. 13–23.

[MFD08] T. Mens, J. Fernández-Ramil, and S. Degrandsart, "The evolution of eclipse", in *Proceedings of the International Conference on Software Maintenance*, IEEE, 2008, pp. 386–395.

[MKI$^+$03] R. Milo, N. Kashtan, S. Itzkovitz, M. E. Newman, and U. Alon, "On the uniform generation of random graphs with prescribed degree sequences", *ArXiv preprint*, 2003.

[MFH00] A. Mockus, R. T. Fielding, and J. Herbsleb, “A case study of open source software development: The Apache server”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2000, pp. 263–272.

[Moc10] A. Mockus, “Organizational volatility and its effects on software defects”, in *Proceeding of the International Symposium on Foundations of Software Engineering*, ACM, 2010, pp. 117–126.

[MFH02] A. Mockus, R. T. Fielding, and J. D. Herbsleb, “Two case studies of open source software development: Apache and Mozilla”, *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, 2002.

[MTW93] H. A. Müller, S. R. Tilley, and K. Wong, “Understanding software systems using reverse engineering technology perspectives from the Rigi project”, in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press, 1993, pp. 217–226.

[NMB08] N. Nagappan, B. Murphy, and V. Basili, “The influence of organizational structure on software quality: An empirical case study”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2008, pp. 521–530.

[NYN$^+$02] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, “Evolution patterns of open-source software systems and communities”, in *Proceedings of the International Workshop on Principles of Software Evolution*, ACM, 2002, pp. 76–85.

[NR69] P. Naur and B. Randell, “Software engineering: Report of a conference sponsored by the Nato science committee”, *Scientific Affairs Division*, 1969.

[NBW06] M. Newman, A.-L. Barabasi, and D. J. Watts, *The structure and dynamics of networks.* Princeton University Press, 2006.

[OJ07] W. Oh and S. Jeon, “Membership herding and network stability in the open source community: The Ising perspective”, *Management Science*, vol. 53, no. 7, pp. 1086–1101, 2007.

[OSdO$^+$12] G. A. Oliva, F. W. Santana, K. C. M. de Oliveira, C. R. B. de Souza, and M. A. Gerosa, “Characterizing key developers: A case study with Apache Ant”, in *Proceedings of the International Conference on Collaboration and Technology*, Springer, 2012, pp. 97–112.

[ORM03] P. Ovaska, M. Rossi, and P. Marttiin, “Architecture as a coordination tool in multi-site software development”, *Software Process: Improvement and Practice*, vol. 8, no. 4, pp. 233–247, 2003.

[PBMW99] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web.”, Technical Report 1999-66, 1999.

[PBD$^+$14] S. Panichella, G. Bavota, M. Di Penta, G. Canfora, and G. Antoniol, “How developers’ collaborations identified from different sources tell us about code changes”, in *International Conference on Software Maintenance and Evolution*, IEEE, 2014, pp. 251–260.

[PCDO14] S. Panichella, G. Canfora, M. Di Penta, and R. Oliveto, “How the evolution of emerging collaborations relates to code changes: An empirical study”, in *Proceedings of the International Conference on Program Comprehension*, ACM, 2014, pp. 177–188.

[Par72] D. L. Parnas, “On the criteria to be used in decomposing systems into modules”, *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[PDS$^+$12] R. Patro, G. Duggal, E. Sefer, H. Wang, D. Filippova, and C. Kingsford, “The missing models: A data-driven approach for learning how networks grow”, in *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, ACM, 2012, pp. 42–50.

[PBD08] D. S. Pattison, C. A. Bird, and P. T. Devanbu, “Talk and work: A preliminary report”, in *Proceedings of the International Working Conference on Mining Software Repositories*, ACM, 2008, pp. 113–116.

[POM10] A. Perianes-Rodríguez, C. Olmeda-Gómez, and F. Moya-Anegón, “Detecting, identifying and visualizing research groups in co-authorship networks”, *Scientometrics*, vol. 82, no. 2, pp. 307–319, 2010.

[RB03] E. Ravasz and A.-L. Barabási, “Hierarchical organization in complex networks”, *Physical Review E*, vol. 67, 2 2003.

[Ray99] E. Raymond, “The cathedral and the bazaar”, *Knowledge, Technology and Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[RKS12] D. Riehle, C. Kolassa, and M. A. Salim, “Developer belief vs. reality: The case of the commit size distribution.”, in *Software Engineering*, 2012, pp. 59–70.

[RGH09] G. Robles, J. Gonzalez-Barahona, and I. Herraiz, “Evolution of the core team of developers in libre software projects”, in *Proceedings of the International Working Conference on Mining Software Repositories*, IEEE, 2009, pp. 167–170.

[RG06] G. Robles and J. M. Gonzalez-Barahona, “Contributor turnover in libre software projects”, in *Open Source Systems*, Springer, 2006, pp. 273–286.

[SMWH09] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, “Tesseract: Interactive visual exploration of socio-technical relationships in software development”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2009, pp. 23–33.

[Sch07] S. E. Schaeffer, “Graph clustering”, *Computer Science Review*, vol. 1, no. 1, pp. 27–64, 2007.

[SLW12] A. Schilling, S. Laumer, and T. Weitzel, “Who will remain? An evaluation of actual person-job and person-team fit to predict developer retention in FLOSS projects”, in *Proceedings of the International Conference on System Sciences*, IEEE, 2012.

[SMS15] I. Scholtes, P. Mavrodiev, and F. Schweitzer, “From Aristotle to Ringelmann: A large-scale analysis of team productivity and coordination in open source software projects”, *Empirical Software Engineering*, pp. 1–42, 2015.

[SSS07] F. Shull, J. Singer, and D. I. Sjøberg, “Guide to advanced empirical software engineering”, in, Springer, 2007, pp. 312–336.

[SCC$^+$12] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. Mcdermid, and R. Paige, “Large-scale complex it systems”, *Communications of the ACM*, vol. 55, no. 7, pp. 71–77, 2012.

[SER04] M. E. Sosa, S. D. Eppinger, and C. M. Rowles, “The misalignment of product architecture and organizational structure in complex product development”, *Management Science*, vol. 50, no. 12, pp. 1674–1689, 2004.

[SL08] D. Spinellis and P. Louridas, “The collaborative organization of knowledge”, *Communications of the ACM*, vol. 51, no. 8, pp. 68–73, 2008.

[SGCH01] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design", in *Proceedings of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM, 2001, pp. 99–108.

[TRC10] A. Terceiro, L. R. Rios, and C. Chavez, "An empirical study on the structural complexity introduced by core and peripheral developers in free software projects", in *Proceeding of the Brazilian Symposium on Software Engineering*, IEEE, 2010, pp. 21–29.

[TMB10] S. L. Toral, M. R. Martínez-Torres, and F. Barrero, "Analysis of virtual communities supporting OSS projects using social network analysis", *Information and Software Technology*, vol. 52, no. 3, pp. 296–303, 2010.

[Tor] L. Torvalds, *Git*, `https://git-scm.com/`, Accessed: 2016-04-26.

[TW02] C. A. Tracy and H. Widom, "Distribution functions for largest eigenvalues and their applications", Higher Education Press, 2002, pp. 587–596.

[vHee] D. van Heesch, *Doxygen*, `http://www.stack.nl/~dimitri/doxygen/`, Accessed: 2016-04-25.

[WOB08] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models", *Empirical Software Engineering*, vol. 13, no. 5, pp. 539–559, 2008.

[WSDN09] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2009, pp. 1–11.

[XCK14] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: A new form of architecture insight", in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2014, pp. 967–977.

[XF14] Q. Xuan and V. Filkov, "Building it together: Synchronous development in OSS", in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2014, pp. 222–233.

[YK03] Y. Ye and K. Kishida, "Toward an understanding of the motivation open source software developers", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2003, pp. 419–429.

[YBH12] Y. Yu, A. Benlian, and T. Hess, “An empirical study of volunteer members’ perceived turnover in open source software projects”, in *Proceedings of the International Conference on System Sciences*, IEEE, 2012, pp. 3396–3405.

[ZMN15] X. Zhang, T. Martin, and M. E. J. Newman, “Identification of core-periphery structure in networks”, *Physical Review E*, vol. 91, 3 2015.

APPENDIX A

A.1 Developer Communities Questionnaire Solicitation E-mail

Dear Open-source developer,

The University of Passau is hosting a survey to evaluate the use of data in the version control system to quantitatively and accurately infer social relationships, collaborative patterns and community structures in open-source projects. You have been selected for this survey because of your contributions to [project name] during the development of [revision]. By participating, you are contributing towards a better understanding of the important social and collaborative factors that lead to prosperity in open-source software development. You will also be able to see the community structure that you participate in and have a chance to comment on it's accuracy. All your information will be kept confidential. We have no interest in judging you as a person, we are merely interested in learning about your experience as an open-source developer.

The survey is composed of 8 questions and should take around 10-15 minutes to complete. Each and every survey response is important to us and we sincerely hope you will take the time to participate. Upon completion of the survey you may include your return e-mail address so that we can send you the survey results.

We thank you for your time and effort.

A.2 Analysis Window Selection

We chose to use a sliding-window approach in our study to generate the time-resolved series of developer networks. Another option would have been to analyze the project using non-overlapping windows, but this can lead to problematic edge discontinuities between the analysis windows. For example, a set of several related changes to the software could be divided between two different analysis windows, even though the changes occurred temporally close together. For this reason, a sliding-window approach superior to the alternative for our purposes, but we also recognized that overlapping windows could influence the appearance of developer transitions (see Section 5.2.2.2), because a commit can appear in two contiguous analysis windows. To test whether the overlapping windows distorts the overall outcome, we compared all Markov chains using non-overlapping windows with those using overlapping windows. The comparison revealed that, in all projects, our conclusion that core developers are more stable than peripheral developers is true regardless of which windowing strategy is used. In most cases, using non-overlapping windows increased the probability that a core or peripheral developer leaves the project, but peripheral developers are always significantly more likely to leave. For example, in QEMU, using overlapping windows, core and peripheral developers leave a project with 0.5% and 10.9% chance respectively. In the case of non-overlapping window this changes to 13% chance for core and 55% chance for peripheral.

A.3 Semantic Coupling

To determine function-level semantic coupling, we first extracted the implementation for each function in the system, including all source code and comments. We then employed well-established text-mining preprocessing operations with minor modifications for our specific domain requirements. In this framework, each function is treated as a "document" in the text-mining sense of the word, and then the document collection was processed use the following processing operations.

Preprocessing The preprocessing stage primarily focuses on reducing word diversity and elimination of words that contain little information. *Stemming* is to used to reduce words to their root form by removing suffixes (e.g., "ing", "ly", "er", etc.) from each word in the document. Stemming is necessary because, even though a root word may have several forms by adding suffixes, it typically refers to a relatively similar concept in all forms. In software engineering, there is a number of variable-naming conventions, such as letter-case separated (e.g., CamelCase) or delimiter separated words that need to be tokenized appropriately. We added additional preprocessing stages to specifically handle proper tokenization of popular naming conventions. For example, the function identifier "get_user" or "getUser" are separated into the two words "get" and "user". One simple example of why this is important is that getters and setters interacting with the same attribute would be incorrectly understood as distinct concepts without appreciating the variable-naming conventions. The final stage of the preprocessing is to remove words that are known not to contain useful information based on a-priori knowledge of the language. For example, words such as "the" are not helpful in determining the domain concept of a document. Removing these words is beneficial for the computational complexity and results by reducing the problem's dimentionality and attenuating noise in the data.

Term Weighting After the preprocessing stage, we arrange all remaining data into a term–document matrix, for mathematical convenience. A term–document matrix is an $M \times N$ matrix with rows representing terms and columns representing documents. For example, an element of the term–document matrix $TD_{i,j}$ is non-zero when document d_j contains term t_i. All elements of the term–document matrix are integer weights that indicate the frequency of occurrence of a given term in a given document. We then apply a weight transformation to the term–document matrix based on the statistics of occurrence for each term. Intuition suggests that not all terms in a document are equally important with regard to identifying the domain concept. The goal of the weighting transformation is to increase the influence of terms that help to identify distinct concepts and decrease the influence of the remaining terms. The particular weighting scheme we applied is called *term frequency-inverse document frequency*:

$$\textit{tf-idf}_{t,d} = \textit{tf}_t \times \log \frac{N}{\textit{df}_t}. \tag{A.1}$$

The term $\textit{tf}_t$ represents the global term frequency across all documents. The second term is the logarithm of the inverse document frequency, where N is the number of documents in the total collection and $\textit{df}_t$ is the number of documents that term t appears. Upon closer inspection, one can recognize that Equation A.1 is: (a) greatest when a term is very frequent, but only appears in a small number of documents, (b) lowest when a term is present in all documents, and (c) between these two extreme cases when a term is infrequent in one document or occurs in many documents.

Latent Semantic Indexing Even for a modest-sized software project, the number of terms used in the implementation vocabulary easily exceeds the thousands. The problem with this becomes evident when adopting the vector-space model, where we consider a document as a vector that exists in a space spanned by the terms that comprise the document collection. Fortunately, this very high dimensional space is extremely sparse, which allows us to project the documents into a lower dimensional subspace which makes the semantic similarity computation tractable. We achieve this using a matrix decomposition technique that relies on the singular value decomposition called *latent semantic indexing*. An added benefit of this technique is that it is capable of correctly resolving the relationships of synonymy and polysemy in natural language [BR+99]. Furthermore, latent semantic indexing has shown evidence to be valid and reliable in the software-engineering domain [BDO+13].

Semantic Similarity In the final step of the analysis, we determine semantic coupling by computing the similarity between all document vectors projected onto the lower dimensional subspace attained from applying latent semantic indexing. We operationalize the similarity between two document vectors in the latent space using cosine similarity

$$\text{similarity}(\vec{d_a}, \vec{d_b}) = \frac{\vec{d_a} \cdot \vec{d_b}}{\left\|\vec{d_a}\right\| \left\|\vec{d_b}\right\|}, \tag{A.2}$$

where the numerator is the dot product between the two document vectors and the denominator is the multiplication of the magnitude of the two document vectors. Intuitively, cosine similarity expresses the difference in the angle between the two document vectors; it equals 1, when the two vectors are parallel, and 0, if they are orthogonal. Two source-code artifacts are then considered to be semantically coupled if the cosine similarity exceeds a given threshold. We experimented extensively with a number of thresholds by manually inspecting the results and judging whether the functions were, in fact, semantically related using architectural knowledge of a well-known project. We found that a threshold of 0.65 was able to identify most semantic relationships with only a very small number of false positives. We did, however, cautiously chose the threshold to optimize to avoid false positives rather than false negatives.