Individuals in Product Development
Interactions with Teams and Products

By
João Nuno Lopes Castro

Licenciatura in Electrical Engineering and Computer Science, University of Porto, 2000

Submitted to the Engineering Systems Division
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Engineering Systems
at the
Massachusetts Institute of Technology

August 2010

Signature of Author...
Engineering Systems Division
July, 2010

Certified by...
Warren P. Seering, Ph.D.
Weber-Shaughness Professor of Mechanical Engineering and Engineering Systems
Thesis Supervisor

Certified by...
Christopher L. Magee, Ph.D.
Professor of the Practice of Mechanical Engineering and Engineering Systems

Certified by...
Eric Rebentisch, Ph.D.
Research Associate, Lean Advancemente Initiative

Certified by...
Alan D. MacCormack, Ph.D.
Visiting Associate Professor of Technological Innovation, Entrepreneurship, and Strategic Management

Accepted by ....
Nancy Leveson, Ph.D.
Professor of Engineering Systems
Chair, Engineering Systems Division Education Committee

(this page intentionally left blank)

This dissertation is dedicated to Ana, Alberto and Andreia who got me here.

And for Tomás, wherever his dreams might take him.

(This page intentionally left blank)

Em memória da Maria Francisca

18 Julho 2007 – 17 Agosto 2010

(This page intentionally left blank)

Individuals in Product Development

Interactions with Teams and Products

By
João Nuno Lopes Castro

Abstract

This dissertation focuses on how individuals involved in complex product development operate and interact with other people in the project and how they perceive and modify the product. Complex product development requires the collaboration of multiple individuals who are specialists in different disciplines. One of the challenges with the execution of design and development projects is coordinating the contributions of each individual to guarantee an aligned, seamless fit.

I review a selection of the literature on team frameworks, coordination methods and empirical product development studies which address teams, individuals and product architectures and structures.

I then conduct two studies. One focuses on individual to individual communication requirement stability and the other on individual interaction with product structure over the development period. In the first, I examine how the most important communication channels between individuals in multifunctional teams compare across thirteen different projects. In this study I found a direct correlation between functionally similar projects and their network of important communication links between individuals. This indicates that when faced with a problem of similar nature the profile of connections between individuals – which ones are more or less important – will also be similar.

In the second, I study how individuals interact with the structure of a product in four software development projects. I found that most individual work is localized and consists of internal improvement work. When work is done that requires simultaneous modifications of several components, I found that the associations made between components does not follow the existing structural dependencies as indicated by the function calls between components. This behavior is consistent throughout the development of the projects and is not dependent on the design state of the product. The associations made between components are also not a good indicator of future structural dependencies. These observations do not follow the indications from previous work on team interactions and product structure, revealing that individuals make associations beyond those suggested by just the structural connections.

It was also observed that individuals are able to identify and work on the most important components in a product and that work is conducted on components irrespective of their age in the system.

Finally, a real-time observation of project execution method is proposed based on the several analysis steps developed within this thesis. The use of this method can be advantageous for practitioners to verify the progress of project and control deviations from plan.
This thesis contributes directly to the stream of research of coordination in product development and contributes to the practice with new methods to help those involved in large-scale complex product development filter the extensive work done by many individuals and find areas of possible intervention.

Thesis supervisor:	Warren P. Seering, Ph.D
Thesis committee:	Eric Rebentisch, Ph.D.
	Christopher L. Magee, Ph.D.
	Alan D. MacCormack, Ph.D.

Acknowledgements

The work included in this dissertation would not have been possible without the generous support of many people, professors, professionals, fellows, friends and family. This page has been the hardest to write because it seems impossible to do justice to all they have done to help me in so very few words.

I would like to thank Warren Seering, my thesis advisor, for always being available with the very best feedback. He has a knack for knowing how much to say to incentivize you when things are not looking good and especially how to tune you down that bit when your expectations are a little too high. He was impeccable in managing to meet weekly and then spend innumerable hours talking to me as a mentor and friend. I believe that looking back many years from now I will remember MIT most by what I learnt with all the time spent with Warren.

I would also like to thank my committee members Eric Rebentisch, Chris Magee and Alan MacCormack for sharing with me their time and knowledge, patiently giving me feedback, helping me make the research stronger and for guiding me to find the answers to my own questions and teaching me how to walk at the level of the research done at MIT.

I would also like to thank José Santos who has been almost an informal advisor, listening to my ideas and giving me some of the most constructive feedback and challenges to improve my work. I would like to thank Manuel Heitor who was one of the first who believed I could be successful at MIT along with Diogo Vasconcelos and Pedro Guedes de Oliveira.

Thank you to the entire ESD community, most notably Dan Roos, Richard de Neufville and Joe Sussman for inspiring me. Thank you to everyone in the Lean Advancement Initiative, especially Debbie Nightingale, Tom Shields and Lissa Natkin who were eternally patient and helpful in navigating bureaucracy.

Thank you to Mark Avnet for kindly providing me with the data for one of the studies in this thesis and thank you to all the companies who allowed me to visit and the employees who spent their time talking to me.

A big thank you to my friends and fellow research group colleagues, Sid Rupani, Dan Livengood, Dan Gillespie, Dave Long, Robb Wirthlin, Pedzi Makumbe and also our external visitors Claudia Wagner, Marcus Pessoa, Christian Briegel and Martin Steinert. Thank you to Aidan Crook for the many hours just hearing my ideas and all the stimulating brainstorms we had walking down the infinite corridor.

Thank you to the communities in the Engineering Systems Student Society, the Portuguese American Post-Graduate Society, the Euroclub and MIT Intercollegiate Volleyball Club for all the fun and friendship.

A big “abraço” and “obrigado” to Alexandre Santos and the gang back in Porto: Tiago Nobre, Vasco Teles and Carlos Carvalho.
My parents, Ana Lopes and Alberto Castro, I could never thank you enough. You’ve been an inspiration and have given me everything. You made me the person I am and gave me the dreams and the tools to fulfill them.
Andreia Rosa, my guilty partner in this adventure and companion for life. My name might be on the cover and I got to put the words on paper but you’re the unspoken author, the person without whose daily encouragement, support and love none of this would have even taken off the ground.
Finally, to the little Tomás who fills me daily with indescribable joy.

To all of you: “Biba!”

Table of contents

List of figures

List of tables

1. Introduction and overview

1.1 Research Motivation

1.1.1 Economic impact of Product Development

Economists divide activities into three major sectors: agriculture, industry and services. The manufacturing, or secondary sector of economic activity, is responsible for roughly 25% of the total economy output and wealth created in the top world economies (table 1).

Economy	GDP	Agriculture (%)	Industry (%)	Services (%)
EU	$16.18 tril	1.9	25.2	72.8
USA	$14.43 tril	1.2	21.9	76.9
Japan	$5.11 tril	1.6	21.9	76.5
China	$4.81 tril	10.6	46.8	42.6

Table 1 – Economic activity by sector. Source: CIA World Factbook 2009

Product development is an integral part of the industry and service sectors. The importance of achieving a successful product development is motivated by the economic impact that it can have. This has lead many authors to focus on diverse aspects of the activity such as methods to correctly capture and understand customer requirements, aid in the generation of new product ideas, scheduling and financing strategies and organizational construction and performance, among others. All of these try to improve one of the elements of the typical performance triad: cost, schedule and quality.

1.1.2 Product development projects

With complex products and high output industries, product development is no longer a single action event or a process which can be accomplished by only one person (Ancona and Caldwell, 2003). The lifecycle of a product is complex and a series of actions are necessary to deliver value to the customer. There are several stages through which the design and development of a product flows and these typically consist of identifying market opportunity, capturing user needs, expressing requirements, detail design and testing which are followed by production, sales, costumer support and end of life treatment. For each stage different people with different skills are required, creating an organizational challenge in the attempt to have all actions coordinated. (Drucker 1988) Also, most design actions in product development have ramifications that impact other parts of the product, people working on the product or the development process itself (Giffin et al. 2009). This poses one of the most serious challenges for efficient and effective product development.
Clark and Fujimoto summarized this best when they wrote:

> "In a turbulent, competitive environment in which customers are demanding and speed is essential, the underlying source of superior performance is integration."
> "Integration means linking problem-solving cycles, bringing functional groups into close working relationships, and achieving a meeting of the minds in concept, strategy, and execution." (Clark & Fujimoto 1991)

A product development organization that is able to correctly align all the contributions and solve all the conflicts in a design process and meet the products' requirements will have a competitive advantage over those that don't. Those in charge of these projects have that responsibility (Deming 2000a) and, while researchers spent time studying how to improve the coordination of teams and the performance of product development organizations, many projects still fail or are delayed with severe cost and other consequences.
The challenge is a daunting one. Large, complex products have thousands of components. Organizations have thousands of technically skilled people. All components in a product are included for a purpose and they interact with other components establishing relationships which have to be managed. People contribute to the product with the knowledge and the solutions for which they are trained. A change in one component may imply a chain reaction of changes across other components in order to accommodate the first one. One specialist's objective function may collide with another and tradeoffs must be determined.
There are a myriad of issues to be addressed in a product development project, be they between technical people, between product components and between people and components.

A better understanding of how people interact with people and the product can lead to better product development. That is the aim of this thesis.

> “I asked him what advice he would give himself if he could time travel back to the start of the project. “It's a twenty-person project," he replied. "It's not a one- or two- or three-person project, and it's not three hundred people. The ambition requires a medium-size team, and when you have a medium-size team, that comes with consequences. There's a lot of coordination. We're still learning, and we're going to continue to learn. How do you execute? How do you set goals? Because I had not done this kind of hands-on twenty-person project with this scope of ambition before, there are so many things that have been surprising and continue to be a surprise.”
> Scott Rosenberg interviewing Mitch Kapor about the Chandler project (Rosenberg 2008)

1.2 Research objective and approach

The objective of this thesis is to develop a better understanding on how product development project participants interact with one another, namely the consistency of interactions among them in different projects, and how they interact with the product they are designing, namely their choice of different product components to make improvements or solve problems. The secondary objective is to translate the findings for practitioners and to develop recommendations for managers of product development projects.

Since the study subject is imminently in a practitioner’s realm and that is where objective data comes from, the work in this thesis will be founded on extensive field observations.

It is the nature of design, invention and product development that the same process can never be repeated, because it cannot be “un-invented” or “un-designed”. In contrast, manufacturing can typically produce many times the same component or assemblies and it is possible to compare between different procedures. In product development that is difficult but there are nevertheless different steps that can be taken and lessons that are learned from other projects. This thesis studies over 30 different projects, some in more depth than others.

This work relies heavily on the individual perspectives of those involved in their product development projects. Taichii Ohno, credited as the person who helped Toyota invent Lean Manufacturing, thought that “assembly workers could probably do most of the functions of the specialists and do them much better because of their direct acquaintance with conditions on the line.” (Womack, Jones & Roos 1990) It is this personal and direct perspective of those involved in the development of new products that this thesis focuses and relies on for its studies.

Interactions of individuals with other teams

One set of activities that this thesis looks at are the interactions between the different project participants. Individuals in product development projects interact with one another to find solutions and coordinate actions. When projects are staffed with many people, the possibility for all of them to communicate with every one else is very slim so a prioritization of which connections will happen and how to set up the organization that facilitates the necessary connections to happen is addressed by this thesis.

Interactions of individuals with the product

Another set of activities that this thesis focuses on are the interactions between the project participants and the product they are designing. If in complex products it is required to have many differently skilled individuals, then they will also have different perspectives regarding the product. How they act and react to the changes they observe on the product is addressed in another section of this thesis.

1.3 Overview of the study

This thesis is comprised of three main sections. The first is a review of literature on the topic of organizations, coordination and product development. The second is a study on the people-to-people interaction in projects while the third section focuses on the people-to-product relationship. Each section introduces the concepts and methods appropriate for the studies within them and, except for the review of state of the art on the topic presented in the literature chapter, they can be read independently.

1.4 Potential Impact of Contribution

This thesis aims to firstly contribute to the academic literature on product development and more precisely coordination within product development projects. Secondly, and since the nature of the study subjects are all applied, this thesis aims to provide new insights or methods based on its findings for practitioners of product development.

2. Review of the state of art

2.1 Literature review

Organizations are created when there is an objective that can't be easily met by a single person. When people get together they establish a set of expectations and distribution of duties such that the collective objective can be reached. How to best set up these organizations and how they operate have always been a concern. Early societal rules established the operational norms and then laws by which people operated. Military command and control structures attempted to mechanize as much the role of individuals in some positions but the optimal aggregation of inputs, contributions and behaviors was always challenging. In the modern scientific era many researchers have dedicated their efforts to understanding how organizations and people within organizations can best be structured so that the returns can be obtained with the least amount of effort.

This thesis is also about product development, a specific type of organizational activity. This section of the thesis reviews some of the previous work conducted on the subject of teams and where it applies to the realm of product development.

2.1.1 Product development process

Product development is a process through which an individual or an organization takes an idea and materializes it into a product or service. The different steps which a product goes through until it is ready to be used are not always the same and there are no clear borders between them. Typically a product is first conceptualized and then designed in detail. This design is then tested and optimized before the final solution which is then validated against the projects objectives. The process is not

linear as each step generates new information that may interfere with previous decisions and require repeating previous steps. For example, if during the prototyping phase it is determined that a specific component is extremely expensive to manufacture or prone to failure, then the design may be altered and find an alternate solution that avoids the use of that component.
The distinction between stages is also not always clear and depends on the methods and process in use. Several authors have identified different stages in the product development process and I have also been able to make empirical observations through case-studies where this process is viewed as a discrete succession of different phases. All authors try to define clear boundaries between activities necessary to product development but, as is shown in the following figure, those borders rarely coincide across authors.

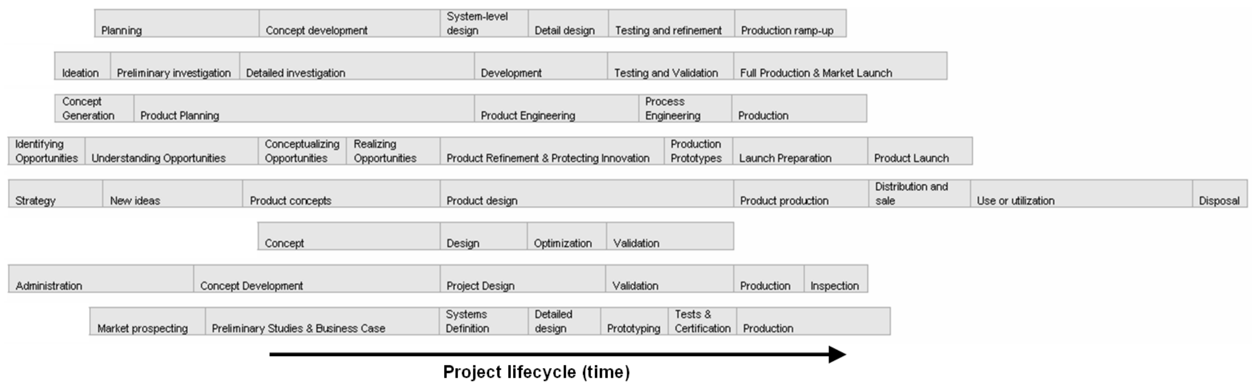

Planning | Concept development | System-level design | Detail design | Testing and refinement | Production ramp-up
Ideation | Preliminary investigation | Detailed investigation | Development | Testing and Validation | Full Production & Market Launch
Concept Generation | Product Planning | Product Engineering | Process Engineering | Production
Identifying Opportunities | Understanding Opportunities | Conceptualizing Opportunities | Realizing Opportunities | Product Refinement & Protecting Innovation | Production Prototypes | Launch Preparation | Product Launch
Strategy | New ideas | Product concepts | Product design | Product production | Distribution and sale | Use or utilization | Disposal
Concept | Design | Optimization | Validation
Administration | Concept Development | Project Design | Validation | Production | Inspection
Market prospecting | Preliminary Studies & Business Case | Systems Definition | Detailed design | Prototyping | Tests & Certification | Production
Project lifecycle (time)

Figure 1 - Stages in Product Development processes according to different authors
Each line is a representation of the stages according to the following authors, respectively: (Ulrich & Eppinger 2000), (Cooper 1993), (Clark & Fujimoto 1991), (Cagan & Vogel 2002), European Ecological PD Framework, (Creveling, Slutsky & Antis 2003) and the last two originate from two companies visited in the early stages of this research.

The first difference between models is their scope in terms of lifecycle. Some authors constrain the product development activities to just the core design stages while others describe the whole lifecycle depending on the type of analysis being made. But the most interesting aspect of these different representations of the same process is that the borders between stages are placed in different positions, if at all. The discrepancy between models does not mean that one of them is less adequate than another, only that each author identifies boundaries between tasks in the product development process and places them at different phases of the process. The only transition which all authors seem to agree on is the transition from development to production. By showing these different interpretations I wish to illustrate that the product development process is in fact a continuous activity -a flow- with no clear boundaries between actions, although very different activities take place at different times.

One further example of product development as a flow of activities appears in the study of the Toyota product development system (Morgan & Liker 2006). In this study, the authors present a thorough review of the methods, the culture and the motivations behind one of the most successful companies in PD without ever having to present a multi-stage representation of the process.

2.1.2 Review of selected team literature

Each task in the development process requires a set of specialized skills and in the development of complex products this typically means requiring teams of multiple specialists to execute them. Michael Schrage (Schrage 1995) considered there to be three variables that define the existence of organizations: design, breakdown and constraints. The first two establish the requirements, the desires and the needs and the objectives that bring people together. Either there is a vision for a new solution or there is a problem due to something no longer operating as desired. Constraints are all the drag forces that limit the execution of the project and they typically include required expertise, schedule, cost, quality and competition.

His work and others (Schneider & Barsoux 2002; Brett, Behfar & Kern 2006) also address the cultural norms that have impact on how teams operate and their cultural predisposition for hierarchical structures, entrepreneurial spirit or simply a tradition in the engineering and design domains. Despite their differences, in general these all expect a standardized and rote approach to how the collaboration is done.

Schrage then also notes that there is a new approach, which he calls the "ad-hocracies" where people act in a "fluid, organic, and selectively decentralized" way. He praises the high flexibility and adaptability of this approach but also notes that "tossing a few people into a room and nailing them to a deadline does not an ad-hocratic collaboration make" and there are many challenges still unsolved. He concludes by suggesting that a great deal of time should be spent by organizations and "cartographers" simply tracking who talks to whom and what is the outcome of those relationships. This idea is in fact one that had been adopted by many researchers in different cases studies. A compendium of different applications and methodologies (Brannick, Salas & Prince 1997) notes that there are many factors that can be analyzed and the selection of such depends on the purpose of the study. Early organizational performance studies focused on global team outcomes such as the cited bombing accuracy in WWII by Thorndike in 1949. Other studies can focus on communication content, individual behaviors within the context of coordination engagements or the impact of team size on performance (Knowledge @ Wharton 2006). Depending on which factors are selected for different studies and the different settings, applications and constraints that create unique perspectives on the organizations, authors who have studied this subject have created frameworks of understanding through which they try to explain how organizations operate. I review some of these frameworks, namely those that apply most directly to the actions of product development organizations.

2.1.3 Review of selected organizational and team frameworks

To understand relationships between actions in organizational environments, authors typically construct frameworks:

Framework	Author(s)
Coordination as markets	(Malone 1987)
Three Focus Areas	(Clark & Fujimoto 1991)
Matrix Organization	(Cusumano & Kentaro 1998)
X-Teams	(Ancona, Bresman & Kaeufer 2002)
Three Cultures of Management	(Schein 1996)
Boundary Objects	(Carlile & Schoonhoven 2002)
Ten Faces of Innovation	(Kelley & Littman 2005)
Self-emerging Knowledge Based Development	(Kennedy 2003)
Toyota LPDS	(Morgan & Liker 2006)
Scope Diseconomies	(Bresnahan, Greenstein & Henderson 2009)
Agreement Matrix	(Christensen, Marx & Stevenson 2006)

Table 2 - Selected organizational frameworks which include coordination as one of their concerns

Each framework consists of a macro-structure of forces in operation within the organization, and describes current state and improvement strategies according to those forces. These frameworks are simplified models that allow understanding of the inherent complexity of the organization but still manage to focus on more than one aspect of the organization.

As an example of a framework, Clark and Fujimoto (Clark & Fujimoto 1991) described the product development system as being focused on three main aspects: specialization, internal integration and external integration. The first focuses on the individual capabilities of each member of the development team and the execution of individual tasks with speed and efficiency. The second and third objectives focus on achieving fast development through task coordination and matching the product with customer expectations, respectively.

While this is enough to address the challenges relating to the humans involved in the process, it is not enough to distinguish the challenges faced with the products themselves. The complexity of a product will affect the amount of resources required to develop it. Recognizing this issue, the same authors proposed an additional framework to distinguish products according to their complexity in terms of internal product structure and the external user interface. By doing this, they distinguish between the complexity required from the designers to develop a functional product and the complexity as perceived by the user of the product. While the user interface complexity may damage the product's attractiveness in the market and efforts to reduce it should be considered, the internal integration is the one that more closely is affected by the organization trying to bring the product to the market and is the topic on which I will focus.

A product with high internal complexity often requires that people with different skills and areas of expertise work together on the same product and bring to it their individual contributions. Also, the product development cycle requires distinct specialties at different points of the development process.

Another example is the Matrix Organization (Cusumano & Kentaro 1998) by which organizations are staffed based on technical or functional domains and projects are executed by using people from each domain. The intersection of multiple projects across all the different disciplines creates a matrix representing who from which area is assigned to which project. This is useful to understand how staff is being allocated and the different available functional areas and number of projects being developed.

A final example is the Toyota Lean Product Development System (Morgan & Liker 2006) which recognizes many of the challenges of team product development such as information exchange, technology adoption, capturing market needs and, for each, an approach is devised. In the lights of this framework many methods are used, some of them concurrently.

The remaining frameworks listed in Table 2 all more or less follow the same spirit. They describe the organization or their challenges along a set of constructs and rules tying them and place different motivations and actions into compartments which seem to describe the operations of the organizations which adopt it. It is then within these managerial settings that different methods are applied which attempt to engage and guarantee that the different actors within a complex organization are coordinated and their actions align. The next section presents some of the different methods used in product development or organizations trying to coordinate the inputs of many.

2.1.4 Review of selected product development coordination methodologies

At a lower level of detail, several methods have been developed to try to promote better coordination between actors in the product development process. I have divided some of the most popular methods used in industry or often cited in academic research, depending on their major topic focus. The different methods have in common their aim of attempting to guarantee that the information between two actors in a project is efficiently made available to them and at the moment it is required. The strategies by which this is accomplished vary, although some share similarities among them. I have categorized them according to theses strategies:

Centralize People – methods in this category try to guarantee that the two actors which need to interact and coordinate are geographically close to each other. The proximity between them allows serendipitous discovery and interactions so when the need for coordination arises the channel is easy to set up. Drawbacks include, for example, the high cost to maintain large teams very close together, the fact that required expertise may change as the project matures and that it simultaneously promotes less useful information exchanges that may be a distraction.
Examples of methods in this category are:

Core team (Cooper 1993; McGrath 1996),
Co-location (Allen 1984)(Rich & Janos 1996),
"Touch and go" (field observation, described in the next section of this thesis)
Integrated concurrent engineering (Hauptman & Hirji 1999) and
Integrated product team (Gerwin & Barrowman 2002)

Centralize Information – methods in this category aggregate the information generated by different actors in a single location so it can be more easily found when sought and also promotes the broadcasting of information to actors with which there is not a known dependency for them to coordinate. While this may result in much of the information being considered irrelevant to some, and for which a processing cost is paid, in some cases it may be the mechanism that averts a late discovery and problem-solving.
Examples of centralizing information methods are:

Chief engineer (Morgan & Liker 2006)
Obeya room (Morgan & Liker 2006)
Networked project teams (McGrath 2004)
Knowledge management system (Alavi & Leidner 2001)
Planning (Klein & Miller 1999)

Communication Improvement – methods in this category try to facilitate the interaction between all members of the organization. By lowering the cost associated with establishing a communication channel or transmitting information between any two members, coordination events can happen more often. Methods in this category include:

Liaison engineer (Clark & Fujimoto 1991)
Project matrix (Cusumano & Kentaro 1998)
Knowledge sharing (Gershenfeld 2005; Allen 1984)
Extreme programming (Paulk 2001)

Structure Communication – methods in this category try to create settings in which the different participants of the product development project are forced to solve cross-disciplinary issues before the project is allowed to progress. There is a formal control of the communication.
Examples of methods in this category are:

Gate Keeper (Allen 1984)
Meetings

Structured Processes – methods in this category are responsible for managing a development process and establishing milestones in which reviews are conducted and issues are raised and addressed.
Examples of processes which dedicate phases to ensure coordination between teams are:

Spiral development (Boehm & Hansen 2000)
Stage-Gate (McGrath 1996)

While these methods have proven popular among the industry, there is no consensus that any one of them has been selected as the definitive answer to coordinate the actions of individuals in these settings. Companies employ one or more methods they believe are better but continue to search for improvements. Toyota is one of the earlier adopters of new methods, some of them developed internally, and uses several of them simultaneously in their development process with the expectation that one or a conjunction of some are able to address a coordination issue and all issues are addressed (Morgan & Liker 2006). In a survey of 65 projects with a software firm, Kraut (Kraut & Streeter 1995) described the simultaneous use of methods and related the use of each to a perceived value in coordinating people. The study found that the most valuable and frequently used methods were direct communication between peers, milestone reviews and co-location.

2.1.5 Review of selected studies on team and product architecture

An inherent challenge in coordination in product development is that the work is distributed among several individuals and sub-teams. Von Hippel (von Hippel 1990) proposed that tasks could be distributed in such a way that their interdependencies would minimize coordination issues. His first idea consisted of mapping dependencies between tasks by inferring how much new information one task would generate and which other tasks depended on it. Distributions of tasks to different people would then take this into account and allocate them in a way that would minimize the number of interfaces between people. The second idea consisted in then lowering the cost associated with those tasks that required crossing the interface between people. This line of thought was then further developed and operationalized (Braha 2002).

Communication patterns

Even a good task distribution will require at some point communication between two individuals or groups in order to establish coordination. Linus Torvalds, the project leader for the Linux operating system, wrote once that "The gating issue in any large project is pretty much all about (a) getting the top people and (b) communication." (Torvalds 2010). Researchers have not ignored this and while the first does not fit in the scope of this thesis, several have dedicated their time to the second and studying how people communicate within project organizations. A review piece on the state of product development research (Brown & Eisenhardt 1995) considered that one of the three main areas of interest for the field was the "communication web", with the other two being the "rational plan" by which organizations better meet the opportunities in the market and the "disciplined problem solving". One of the most cited works on this issue mapped the frequency with which each individual communicated with others and found there is a direct relationship between this and their physical distance (Allen 1984). Other studies have looked at patterns of communication between disciplines, such as marketing, engineering and manufacturing (Griffin & Hauser 1992) or between research and development, marketing and operations (Olson et al. 2001). At least one study has also taken the separation of work from a functional area perspective and tested the importance of eleven factors for cooperation in new product development (Kim & Kang 2008). Another study analyzed messages exchanged by employees of a large organization and verified that not only spatial distance had an impact on frequency of communication but also the organizational structure (Kleinbaum, Stuart & Tushman 2008).

Other studies have linked the communication channels predicted by those involved at the beginning of a project with what then occurs during the execution (Morelli, Eppinger & Gulati 1995). Some more recent studies of similar nature have incorporated newer communication tools, namely email and found that communication patterns of high-performing teams reveal more messages being exchanged than the rest (Chiocchio 2007).

Team communication and product structure

Since the individuals and teams in product development exchange information related to the product, it is natural that the relationship between both has also been studied. Some authors have inclusively proposed that if there is a design interaction in the product then the teams designing those components must communicate (Lawrence & Lorsch 1967; Allen 1984; Rich & Janos 1996). The relationship between the product structure and what communication channels are activated was empirically tested (Sosa, Eppinger & Rowles 2004) and showed that team interactions most likely would happen where there were product dependencies. The same relationship was established when comparing the structure of the product and the dependencies along the development process and the organizational ties (Eppinger & Salminen 2001).

An early proposal of the direct relationship between the product structure and the organization developing it was made by Conway (Conway 1968) when, in this seminal paper, he states that "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations". This passage has become known as "Conway's law" and is often referred to by practitioners although the amount of academic studies with empirical data to support it is very limited. This limitation is due to the amount of detailed data on the product structure and communication necessary to establish the relationship, in addition to the number of different projects necessary for comparison. One attempt to overcome this was made through a review of academic literature that, while not directly mentioning or addressing the issue, did contain arguments that support it (Colfer & Baldwin 2010).

Another study on the subject (MacCormack, Rusnak & Baldwin 2008) was able to compare six pairs of software projects in which each pair had two different organizational settings styles. These two styles were common to all pairs. The study was also able to analyze in detail the resulting products and found that there was a clear distinction in observations between organizational style and product outcomes. This research takes advantage of previous work on properties of product structures (MacCormack, Rusnak & Baldwin 2006) which reflects on the modular structure of products and their evolution. The methodology used to extract the structure of the product described in this paper is also used in this thesis in chapter 4.

The temporal variation of the performance of teams and their engagement in coordination activities has been the focus of two authors. One paper which analyzed 50 R&D projects within a single organization reported that participants would become less engaged in communication as the projects

evolved and stabilized (Katz 1982). In contrast, a later study of 34 products found the opposite when considering multi-disciplinary groups besides R&D (Olson et al. 2001).
Another study looked at 12 different design sessions and compared the social interactions and the corresponding shared knowledge with the maturity and challenges of the project being carried out (Avnet 2009).

Product development and network analysis

Since communication between individuals establishes linkages between actors it seems natural that an observation of an entire team or organization be represented as a graph. The largest burden in doing this is collecting the data from all the participants (Allen 1984) but newer methods based on the use of communication technologies have allowed researchers to more easily collect and analyze this data. In parallel, the stream of research conducted in network analysis has also gained traction (Wasserman & Faust 1994) and we can now find researchers who have joined the two.
One study of social networks (Klein et al. 2004) found that members with higher education moved to the center of the network and that those who shared similar cultural values would tend to be associated with each other, in line with the general theory of homophily (Ruef, Aldrich & Carter 2003).
Network analysis techniques are also useful to analyze other connected representations such as the task dependencies within product development. In a double report (Braha & Bar-Yam 2004; Braha & Bar-Yam 2004) the authors found a significant difference between tasks that generated information and those that consumed information. Those that generate information exhibit a scale-free connectivity while those that consume information are limited in the number of connections they establish. A more recent study used the same network analysis methodologies and representation of the connections between tasks in product development but instead of exploring common network metrics, it established the relationships between these and how individual elements affected and were affected by their insertion in the network (Collins, Yassine & Borgatti 2008).

Summary

In this section of the thesis I have shown how different researchers have addressed the issue of coordination in product development. Some have established frameworks on the subject so that theories can be tested and others have created or described methods to be used by practitioners. Another part of this review analyzed the different approaches and focus areas that empirical testing has focused on. This showed that coordination within product development can be studied through a lens of the product structure, the development process, the individual communications, the culture and the organizational structure. It also pointed to different methodologies used when studying the subject. Finally, I hope it has also shown how vivid the topic is and how much interest it generates.

2.2 Practitioner review (Case studies)

Product development is very much an industrial activity so, in addition to the review of academic literature, I wished to ground my knowledge on current practices by visiting product development companies.

I set out to get an eclectic mix of examples allowing for variations in industry and market, size of engineering team, geographic location, and work culture. Under the confidentiality agreements, I am unable to identify these companies but they did vary along those characteristics, with teams ranging from about ten to thousands of engineers, located in Europe, North and South America, and developing products for large scale consumer bases, niche specialist markets or industrial costumers. The following sections describe the four most interesting cases visited, with varying degrees of success in their design efforts.

2.2.1 Medical devices

This case illustrates an instance of misalignment between industrial design consultants and mechanical engineers

The company

This company designs and manufactures a small portfolio of equipment used by physicians in hospital operating rooms. The company follows a classic stage-gate process with concept, design, optimization and verification phases. Due to the nature of their product and since a malfunction can cost the life of a person, there is an additional care taken into the verification steps. The company uses a single facility for all its collaborators. This includes management, engineering, manufacturing, sales, service and administrative staff. The portfolio of products simultaneously being commercialized is relatively small and this allows the development team to participate in the design of every product the company produces without having to share resources among products.

What was observed

For the development of a newer model of their leading product, management decided that improvements should be made in the human-factors aspects so that users would be more productive in using the device. This investment was made by contracting a renowned industrial design firm who would be responsible for the outer-shell and interaction, while the company engineers focused on the internal mechanisms.

When the industrial design and the mechanical team attempted to put together their systems, it was verified that the internal components would not fit the designed shell. This case of mis-coordination led to several months of rework and delay in the product launch.

2.2.2 Aerospace equipment

This case illustrates the adoption of a global risk pooling strategy and the use of temporarily co-located teams, a permanent core-team and globally distributed product development.

The company

The aerospace industry is one of the most capital intensive industries of the last few decades. Products such as airplanes and rockets require many years of development and testing before they can be operated or commercialized. An unsuccessful product can cause an entire organization to collapse. Traditional multi-tier industrial organizations relied on lower-tier suppliers who would sell their wares to assemblers and integrators. The commercial relationship between them would exist only at the moment of exchange of goods and the buyer's use of them was of little or no concern for the supplier. Because of this, development organizations have focused on minimizing the risks involved. One of the popular strategies is risk-pooling.

What was observed

The company has been able to develop products with a much lower capital investment and risk for themselves since the risk is shared with its suppliers. The role of the visited company is no longer to procure components and systems and later integrate them into a final product. Its responsibility is to understand the market and to coordinate the network of component and system manufactures in such a way that they can, collaboratively, produce a viable commercial product. Their contribution is to attempt to make the whole more than the sum of its parts and the other participating companies no longer are considered merely suppliers but partners.

The lead organization manages this process by operating a core-team of disciplines which is staffed by people from the organization and by members of different suppliers. Periodically they hold larger gatherings with lead engineers from each partner and their premises can accommodate external staff for extended periods of time.

This case is an example of centralizing people but it isn't as simple as co-location, since it selects which disciplines and few representatives are part of the core-team. It is also frugal as the diverse teams are not required to co-locate permanently. I have previously called this approach "Touch and Go" in the literature review section of this thesis.

2.2.3 Food processing and marketing

This case illustrates the misuse of a common information repository that is tied to the development process in a way that does not allow flexibility. Users don't follow the rigid procedures and miscoordination happens.

The company

This company develops many products every year in the food industry. It transforms raw materials into ready to eat products which are managed under different brands. Each brand then has its range of products which can then be further subdivided into the different ways they get commercialized. An example of a product could be a branded salty snack that gets packaged into snack size portioned bags. Within the company's portfolio it would also be possible to find the same snack portioned differently and within the same brand their might be different flavored snacks. Finally, the company may want to run special campaigns due to seasonality, festivities or any other marketing promotion for which modifications to the base product are made. This company has to manage an immense portfolio of products, with active SKU (stock keeping units) numbers well into the thousands which is typical of this industry.

The range of new products being developed is also immense. Most of the base of products take several years of testing and development in the company's kitchens before they are commercialized but then all the different variants have to be developed. Even if the principal processed ingredient is the same, new packaging to attend a different portion size or marketing campaign creates a new product.

While the company is creating new products, the difference between some of them would be considered trivial in terms of product development. Changing a bag size to accommodate 20% more potato chips is hardly what we would consider 'complex product development' but managing all the different changes at the pace that they do becomes interesting. Since the company goes through the development so many times, a process emerged of all the different required steps that lend it to be comparable to a manufacturing process. While the frequency with which the company repeats the process makes it look like a manufacturing process, the main difference is that product development transforms information, not materials. It is also an uncertain activity, where the outcome is something that is designed and created starting with some premises which may change as development progresses whereas in manufacturing there is a predictable expectation of the outcome.

What was observed

In order to follow the product development processes, the company set up a computerized information tracking tool that allowed members in the process to know in which stage the project is. As each stage completed its work, the information would be introduced into the system and the stages that depended on this as input would be able to commence their work. Over the years, the process suffered changes and the information tool no longer mirrored exactly how the company operated but it was still mostly useful. As a result workarounds appeared.

The most common workaround was to introduce fake data to release a product to proceed to the next step in the process. The system would require a set of fields to have data introduced before work was available to a downstream station but in some projects this information wasn't actually necessary.

The rule employed was to introduce "obviously erroneous data" in the required field. This information would be in the system and available other participants, but it was expected that they would be able to recognize it as invalid numbers. For example, a bag with one ton of candy to be sold to supermarket customers.

This company uses an information centralizing method and a structured process to coordinate their product developers, and has been mostly successful at this. However, since the use of the process has evolved into a scenario where it relies on other members being able to realize what is fake data in the middle of real data it is prone to some mistakes and these have happened, leading to delays and additional costs.

2.2.4 Industrial equipment

This case illustrates how a very small product development team successfully operates in an environment in which coordination and coordination methods are not considered.

The company

This company operates in a niche sector of industrial equipment that processes and transforms heavy raw material. They design and build custom made-to-order equipment which is considered to be the best worldwide in terms of performance and utilization of raw material. Their customers are goods manufacturers all over the world whose products rely on the type of raw materials that this company's equipment processes.

The equipment is designed by three teams: mechanical structure, robotics and software. Mechanical structure is responsible for creating the body of the equipment such that it holds the heavy raw material as it is being processed. The robotics operates the transformative components that operate on the material and the software is responsible for the analysis of the incoming material and what to do with it, minimizing the amount of wasted material.

What was observed

There are typically 9 people in the product development group. One project leader, four with the mechanical structure, two in robotics and two in software. The product development team is small

by typical complex product numbers. While this would allow for the entire team to be co-located, their software development staff is over 250km away from the rest of the organization.
The three founders of the company first met and decided to launch the company where the software team is currently located. A managerial decision was made to locate the shop of the company near its potential clients and so two of the founders migrated towards an industrial zone where they expected to be able to address clients' needs and procure parts more easily. Therefore the company is split up and while it now operates at a global level, there are social roots established by the entire staff that has precluded the two offices from merging geographically.
The managers of the company believe that they have been efficiently managing their operations despite this. They are market leaders in a segment with direct competitors coming from world-class industrialized markets such as Germany and China. When asked about how they achieve coordination of the three teams the reaction was that "we don't do coordination". Nevertheless, the managerial concept of coordinating teams was described as being relevant and they understood its usefulness, but that it wasn't an issue that they spent time on. There had been no problem in the past that they could associate with mis-coordination.
The nature of software as a digital information material and since it is the software team that is remote, has allowed them to maintain the necessary coordination without requiring the overhead of meetings. The software team can digitally transfer code to their physical assembly facility to be tested and the staff there can relay the results back to them. Also, the software team will visit for a day once or twice a month for more detailed inspection of the behavior of the software on the equipment.
In a follow up to the "we don't do coordination" comment, further interviews and data collection were conducted which allowed me to observe that the whole team interacted with one another and exchanged and relied on one another for information. Coordination is happening, but naturally occurring in the environment in which they operate.
This case illustrates that projects with small team sizes and with well known dependencies between the product's modules are able to operate without requiring formal or conscientious coordination.

(this page intentionally left blank)

3. Study I – On the stability of the important coordination channels across different projects

We have seen from a selection of the literature and industrial practice that participants in product development projects need to interact with one another to coordinate the impacts and dependencies of their work on the product.

In this chapter I will analyze what happens in a series of product development projects within the same organization where objectives and the multi-disciplinary teams differ. I will focus on the connections that are made between team members and the nature of the project and relate the two.

3.1 Stability of important coordination channels

As a project is conducted participants may engage other team members in order to solve a mutual dependency or coordinate around a trade-off. Each participant not only has to execute his tasks but is also responsible for navigating the set of possible interactions with other participants.

When doing this, participants should prioritize these connections so that, in case not all of them can be addressed because of resource constraints, at least the more severe are. For very large teams it is unreasonable to expect that a designer will engage with every other single project member in order to verify if there is a dependency and guarantee agreement. The number of total communication links in a project is given by n(n-1)/2 where n is the number of participants in a project.

For any given project, the ability to predict and identify dependencies between different disciplines will allow mangers and project participants to address them. This can have significant impact on project performance and has been the focus of previous studies (Sosa, Eppinger & Rowles 2004; Morelli, Eppinger & Gulati 1995).

When a team takes on several projects, the observed connections among the participants might be similar or distinct when we compare the projects. A connection that was made in one project may or may not be made in another project. The experience acquired in a previous project may tell whether a specific connection is or is not relevant, what issues were identified and, maybe depending on the nature of the product, if the set of connections are or are not different.

In order to test this, I formulate the following hypothesis:

Hypothesis 1 – The communication channels that are relevant in a project will be the same in a different project if these projects are similar

A test of this hypothesis is described in the following sections. First, I explain the project environment requirements and characteristics that are adequate for this test. Next, I describe how data was collected, treated and set up for analysis. I then detail the results and robustness tests. Finally, I summarize the findings and elaborate on how these results may impact practitioners.

3.2 Research setting

3.2.1 Required characteristics

In this section I describe the environment requirements and characteristics that allow the hypothesis to be tested.

The objective framed within the hypothesis is to test the similarity between the communication channels established in one project with other projects done by the same team. This requires a data source where multiple projects have taken place and in which their running conditions are mostly similar. Projects with different scope, budget, schedule, team membership, team size and geographic dispersion among other things can have any one of these factors generating confounding issues into the observations we want to test. While it is almost impossible to control all of them, I found a dataset that is adequately strong for these purposes.

From this dataset we can extract the differences between projects and a measurement of communication between team members.

3.2.2 The NASA Mission Design Center

A recent thesis on socio-technical congruence in design processes collected data at a NASA integrated concurrent engineering mission design center. The resulting document (Avnet 2009) includes a detailed description of this center, including historical developments and its role within NASA and other affiliated design centers. Part of the data that was collected in that study was made available for this thesis for the test of hypothesis 1. In order to familiarize the reader with this design center I summarize or cite the relevant portions of the description from Avnet's thesis.

The center designs spacecraft and mission architecture for Earth-orbiting or planetary missions. This involves the contributions from several disciplines: Attitude Control, Avionics, Communications, Electrical Power, Flight Dynamics, Flight Software, Integration and Test, Launch Vehicles, Mechanical, Mission Operations, Orbital Debris, Parametric Cost, Propulsion, Radiation, Reliability, and Thermal. The design sessions normally involve 20 to 25 people, with one or two representing a discipline, and make up a "full design team working together in the facility throughout the entire design study, which usually lasts about a week". This study looked at 13 different projects in which the team was co-located in one specifically built room with stations for each discipline and the effort required for communication between different disciplines inside this room is negligible. The type of missions the team was asked to design were mostly similar and therefore the "design process [was] somewhat routine" although they did experience some missions that fell out of their "traditional comfort zone". Missions were staffed with a combination of disciplines depending on the design objectives and the people in each position were mostly always the same.

3.3 Data Collection and Analysis setup

In this section I describe how data was collected, treated and set up for analysis.

3.3.1 Data collection

The field data was collected by Avnet through the use of surveys. These surveys were administered online before and after each mission to each mission participant, and an 80% response rate was achieved.
Hypothesis 1 was phrased as: "The communication channels that are relevant in a project will be the same in a different project if these projects are similar"
In order to test this hypothesis we want to be able to observe two things:

1- The communication channels established between project participants and how important they are, and

2 - The differences and similarities between projects

From the survey that was administered to project participants, there was one relevant question for this study. Question number 6 stated:

> "For the current study only, please indicate the importance of direct communication between you, serving in your subsystem role, and each of the other members of the design team. Please use the space below to comment on any particularly interesting or unique design issues discussed with other members of the design team."

The respondent was then asked to report using a four level Likert scale with the following level values:

0 - Unnecessary,
1 - Helpful,
2 - Important,
3 - Essential

Respondents were given a list of all the different functional areas involved in the project in which they participated so they could rate their interactions with the other participants. The differences and similarities between the projects were obtained by analyzing the team constitution. Each project had a team staffed in the required functional areas. Some projects required more or less areas to be staffed depending on the type of mission being designed and so, by looking at how a project was staffed, comparatively to others, we can observe how different they were.
This data was obtained directly by Avnet when setting up the interviews.

3.3.2 Analysis environment

The collected data was transferred onto individual spreadsheets by Avnet representing surveys from each mission. I then treated this data in order to remove any personally identifiable information and removed the portions irrelevant to this study. Subsequently, the data remaining in the spreadsheets was fed into a database system for a more powerful analysis. The entries in the database's main table included:

- the unique, anonymous identifier of the respondent,
- the functional area to which they belonged,
- the project in which the answer was given,
- the target of their observation and
- the value of their response.

By formulating queries to the database it was possible to quickly obtain answers to questions such as:

- Who did person from functional area X consider 'essential' in project 1?
- How many connections considered 'essential' or 'important' were established in project 6?
- Which functional areas considered functional area Y as 'essential' to their work?
- What is the average number of targets considered as 'essential'?

3.4 Data analysis & Results

With the data treated and stored in a query database, I used a mix of software tools such as Mathworks Matlab and Microsoft Excel to calculate and plot the results. In this section I will show the different calculated results and describe how each set was obtained.

3.4.1 Data demographics

The social and technical constraints of how people interact with others in teams is outside the scope of this thesis but an observation of the 41 individuals who participated in these projects reveals that they typically elect a core group of people that are more important to them. On average, participants selected a third of the other functional areas as essential to them (level 3), 39% as useful (level 2) and 28% as only helpful (level 1).

3.4.2 Measuring Project Similarity

The comparison of different projects was done by analyzing which functional areas were staffed in each project. When representing the array of functions used by projects as a binary vector we can obtain a representation of a project as:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1

Table 3 - Functions used by project

Where function names are reduced to a single letter symbol and a "1" denotes that the function is being used for that mission and a "0" denotes that there is no one in that capacity for that specific mission.

The following table is obtained by representing all of the mission vectors:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
Project 1	1			1		1		1	1	1	1	1		1	1	1	1	1	1	1
Project 2	1		1	1		1		1	1	1	1	1	1	1	1	1	1	1	1	1
Project 3	1		1	1		1		1	1	1	1	1	1	1		1	1	1	1	1
Project 4	1			1	1	1		1	1	1		1	1	1	1	1	1	1	1	1
Project 5			1	1	1	1	1	1	1	1		1	1	1		1	1	1	1	1
Project 6	1		1	1	1	1		1	1	1		1	1	1		1	1	1	1	1
Project 7	1		1	1		1						1						1	1	1
Project 8	1		1	1	1	1		1	1	1		1			1	1		1	1	1
Project 9			1	1		1		1	1	1		1	1	1	1			1	1	1
Project 10	1	1		1				1												1
Project 11	1		1	1		1		1	1	1	1	1	1	1	1	1	1	1	1	1
Project 12	1		1	1		1		1	1	1	1	1	1	1	1	1	1	1	1	1
Project 13	1		1		1	1			1	1		1	1	1	1	1	1		1	1

Table 4 - Functions used by all projects

Comparing two projects

In order to verify how similar two projects are I used their respective functional area vectors. The measure of similarity is calculated by obtaining the ratio of the number of functions commonly used by two projects over the number of all functions used by the pair of projects. This is the same as saying in set logic as the ratio of the intersection over the reunion of functions.

For example, comparing the functional vectors of project 1 and 2:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
Project 1	1	0	0	1	0	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1
Project 2	1	0	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 5 - Comparing the functions of two projects

The only difference between these two projects is the use of functional areas C and M. There are 15 functional areas in common and 17 total functional areas. By the measure defined before, these projects have a similarity measure of 15/17= 0.882

Comparing all the projects

By repeating the measurement of similarity for every pair of projects I get:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1													
2	0.882												
3	0.824	0.941											
4	0.824	0.833	0.778										
5	0.632	0.737	0.778	0.778									
6	0.722	0.833	0.882	0.882	0.882								
7	0.438	0.471	0.500	0.412	0.412	0.500							
8	0.706	0.722	0.667	0.765	0.667	0.765	0.571						
9	0.647	0.765	0.706	0.706	0.706	0.706	0.500	0.688					
10	0.250	0.222	0.235	0.235	0.167	0.235	0.300	0.267	0.200				
11	0.882	1.000	0.941	0.833	0.737	0.833	0.471	0.722	0.765	0.222			
12	0.882	1.000	0.941	0.833	0.737	0.833	0.471	0.722	0.765	0.222	1.000		
13	0.611	0.722	0.667	0.765	0.667	0.765	0.375	0.647	0.588	0.118	0.722	0.722	

Table 6 - Correlation coefficient between functional areas of projects

From the analysis of this table we can verify that projects 2, 11 and 12 have the same functional constitution while other vary in their degree of similarity.

3.4.3 Measuring Communication Importance Similarity

The other comparison between projects is their similarity of importance of communication between functions. In each project, respondents graded their peers in terms of importance of communication. As each participant was responsible for a specific functional area, with this information I can establish a correspondence between the different functions and represent it in a table for each project, such as:

Functional Area

Project X

Functional Area

	A	B	C	D	E	F	…
A							
B							
C							
D							
E							
F							
…							

Table 7 - Example of table for communication importance between functions in a project

Comparing two projects

In order to compare how similar two projects were in how their participants regarded the importance to communicate with the other participants in the project, I take the values in Table 7 and represent them as a line by line sequence, obtaining a vector for the importance of communication for a specific project.

When comparing two projects, if they both don't use a function then it is removed from their representations. With two vectors from two projects and by calculating the correlation coefficient between them, I get a measure of how similar the communication patterns in the two projects were.

Comparing all the projects

By repeating the procedure for every pair of projects, I get the full table of similarity of communication importance:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1													
2	0.557												
3	0.430	0.555											
4	0.529	0.564	0.473										
5	0.375	0.446	0.515	0.621									
6	0.448	0.542	0.575	0.631	0.704								
7	0.438	0.401	0.421	0.372	0.368	0.419							
8	0.247	0.301	0.162	0.399	0.372	0.394	0.296						
9	0.401	0.470	0.240	0.388	0.373	0.335	0.421	0.512					
10	0.181	0.132	0.022	0.056	0.038	0.090	-0.069	0.050	0.114				
11	0.423	0.492	0.421	0.482	0.304	0.324	0.327	0.262	0.395	0.012			
12	0.542	0.556	0.417	0.552	0.414	0.397	0.362	0.349	0.441	0.047	0.605		
13	0.273	0.433	0.322	0.501	0.428	0.396	0.261	0.334	0.229	-0.013	0.372	0.487	

Table 8 - Correlation coefficient between communication importance of projects

From the analysis of this table we can verify that the two projects whose importance of communication between participants most resembled each other were project 5 and project 6 and that the projects 10 and 13 were the two least alike.

3.4.4 Joint plot of project similarity and communication link similarity

I can now compare how these two sets of values, project similarity and communication importance similarity, are related. For example, Project pair 1 and 2, have 0.882 in project functional similarity and 0.557 in the importance of communication channels similarity. Graphically this can be represented as a scatter plot, with each marker representing the values for a given pair of projects. Since we have 13 projects, the total number of possible pairs is n(n-1)/2 = 78.

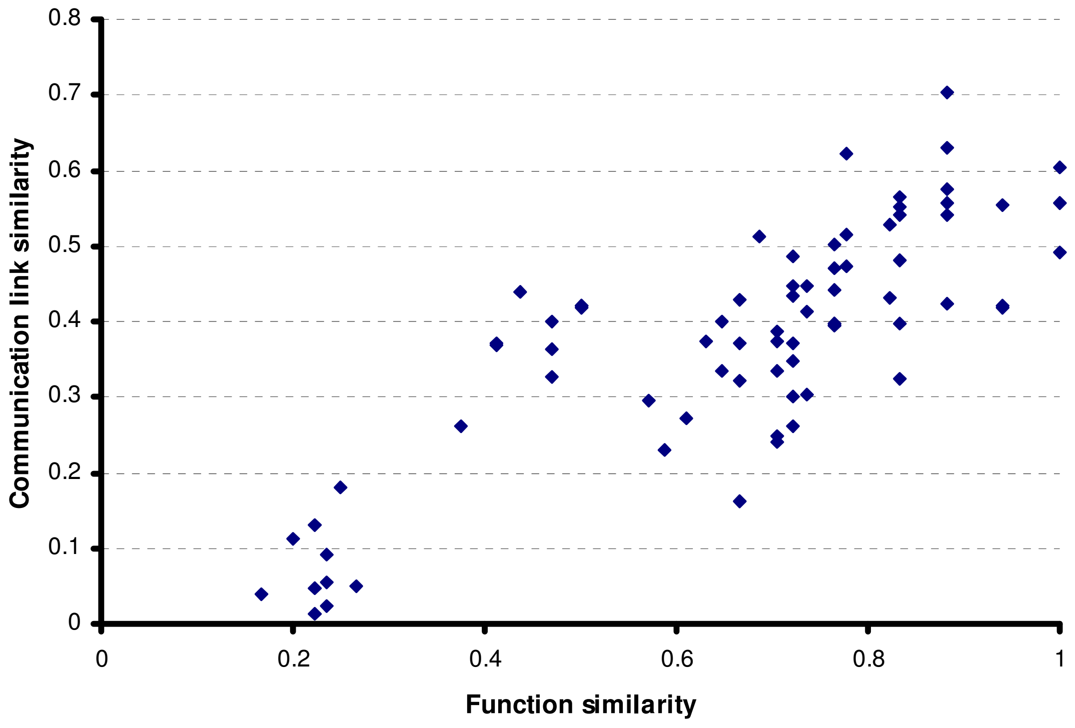


Figure 2 - Relating function similarity and communication importance similarity

From this chart we can immediately observe the correlation between the two measures. Projects that are more similar in their functional domains also exhibit a more similar pattern of importance of communication.

The R^2 value of this dispersion is 0.6818.

3.4.5 Validity test

One of the threats to the validity of these results is auto-correlation between the two observations. Auto-correlation can happen when two different measurement methods reflect the same variable. An example of auto-correlation is using a thermometer in Celsius and another in Fahrenheit. Each will give a different value reading but they are both measuring temperature so a scatter plot will show a clear correlation between the results.

Regarding the measurements being made for these projects, one verifies which functional areas are present in a project while the other measures how important areas in a project regard the other areas in the same project.

The threat of auto-correlation is present since the areas that respond to the second question are the same as those that are present in the project. A functional area that is not present in a project will not provide data on other functional areas.
If, in a reductionist approach, it is considered that all areas communicate with all other areas at the same level of importance then effectively this representation would mirror the functional area data and auto-correlation would be observed. In the actual case, the data collected is much more detailed. Each project participant representing a functional area ranks all other functional areas on a scale. This allows room for differences to emerge between the two types of observations.

Testing auto-correlation

One way to test this auto-correlation is by exploring the solution space of variables. If the importance of communication measurement is constrained, i.e. not independent, from the project similarity measure, then a simulated run of results will also show this.
In each run of the test I assigned to each functional area a random importance value to the other functional areas that were present in a project. I followed the values observed earlier in section 3.4.1 that there is a distribution on the type of connections that are made.
After the connections within each project were established, I computed the similarity value between projects as before. This entire process was repeated ten times, in each run generating a set of random communication importance levels between the functions used by a project and then calculating the similarity values between projects.
This generated (13*12/2) * 10 = 780 values. When plotted together with the observed values we get:

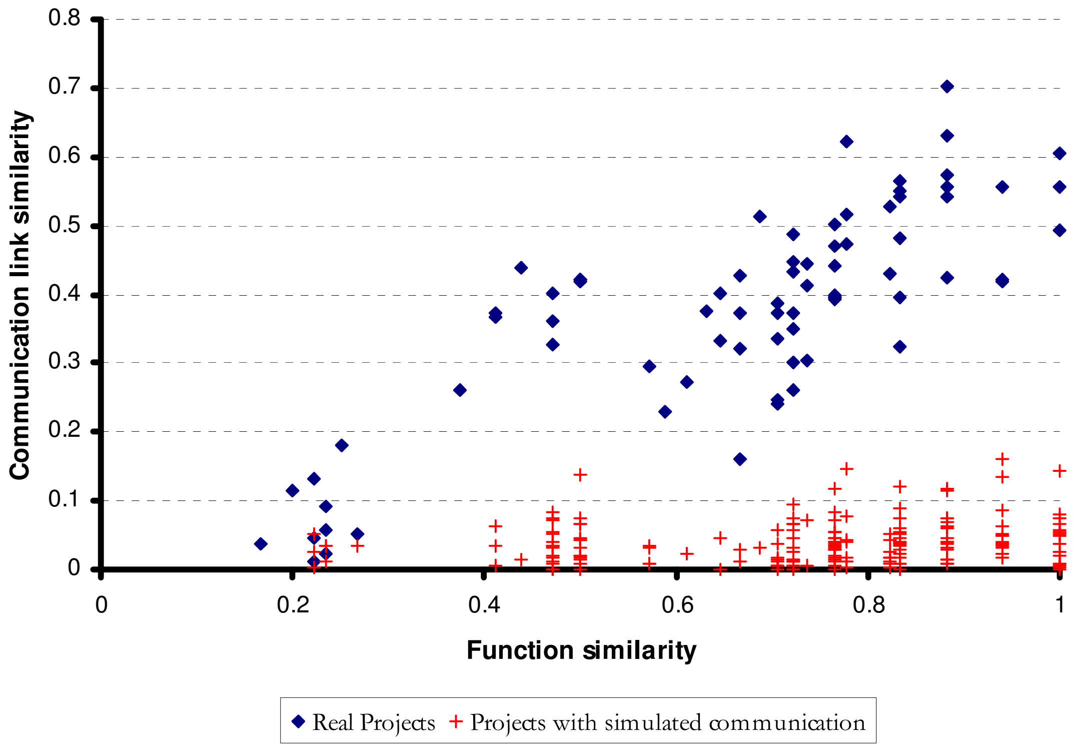


Figure 3 - Function similarity and Communication importance with simulated projects

Of note, this test also illustrates the p-value of the observed results. The p-value is very low, meaning that the observations that were made are very unlikely due to a random assignment of values. Graphically this is seen by how distant the observed values are from those generated in the random runs.

3.5 Conclusion

Hypothesis 1 stated: "The communication channels that are relevant in a project will be the same in a different project if these projects are similar". From these results we can observe that there is a direct correlation between functional similarity and the pattern of important communication channels. It is also observable that this correlation is not simply due to the constitution of the teams. Considering that participants in these projects are free to engage with any other participant, what these results illustrate in terms of team interactions is that similar projects also have similar communication channels being established. When a team starts another project, its participants will mostly interact with the same areas as before given that the new project is similar to the previous one.

Other considerations

The efficiency of the team performance wasn't measured, so it isn't possible to analyze a relationship between the connections being made and the success of the project. It is possible to consider that the number of connections that are being made is, or not, optimal and that there are other sets of connections by which it is possible to achieve the same or better results. Several other projects had been conducted before the data was collected and the participants involved in these programs were experienced and familiar with their role, the design process and the roles of other participants. This can explain the stability of the connections as the solution may be an emergent set of a path-dependent optimization taken by the teams.

Implications for practitioners

The data collected for this study originated from a specific environment. The scope of each project was mission planning and no detailed design was pursued. The duration of each project was limited (1 week) and all participants shared a common space for the duration of the project.

Even simple product development projects have durations that are typically longer than those observed and proceed throughout detailed design and verification stages until entering production. Nevertheless, the observed projects focused on the early stage of a mission design. Most of the cost of a product is decided in the early stages of its development (Ullman 2002) so a study that analyzes only this stage is still relevant.

With a preponderance of similar connections when projects are similar, product development managers can, when setting up new projects, predict with some probability what connections between participants will be important. If a project manager is able to identify all the important connections between project participants then it is possible to optimize the operating environment of that team. The project manager can assign any of the coordination methods to enable the closely related people to exchange the information they require.

By facilitating and focusing on the communications that are important, and avoiding spending effort on the least relevant, practitioners will be able to improve the quality of their product development. This method presents an approach specifying how to identify those connections based on past project experience.

(this page intentionally left blank)

4. Study II – On product structure as a determinant for action in complex product development

We have seen so far some of the complexity present in managing the interactions between participants in product development efforts. In this chapter I will explore how the structure of the product being designed also relates to the work that is conducted. This is an area which has also garnered attention by the research community in the past but has mainly focused on the relationship between the product and the collective social structure, considering teams and the entire organization.

In this chapter I will analyze how individuals, set in a collective product development effort, operate in relation to the product that is being designed. I will first start by detailing the research problem and frame the hypothesis to be tested. With this, I will explore the conditions required to test them. As you will then see, this required me to explore a specific domain in product development for which there is also vast research literature so I conducted a literature review and spent some time researching this domain and describing the most relevant findings. Then, in the data collection and experimental setup I will describe the research methods I used and how I implemented them. Finally, we will be equipped to explore the data analysis and discussion of the results.

4.1 Product structure as a determinant for action

As designers work in a product development project, they introduce changes to a product concept, schematic or prototype in order to build a final solution that meets or exceeds the requirements and specifications. When doing this collaboratively they can discuss and solve tradeoffs in common areas or work on more or less independent components or aspects of a component to then later be assembled. Complex products are characterized by having many interdependent components and the way these components depend on each other or are associated together constitute the product's structure.

When making changes and improvements to a partially designed product, the designer has inherently to take into account the existing components and their relationship with the rest of the product. The designer makes a selection of which components to modify depending on the objective of his change. Changes can be made to a single component or to several of them at the same time to achieve the desired outcome. A change can impact a single component if the nature of the change is independent from the rest of the system. In other cases, a change to one component leads to changes in more components to accommodate the original modification.

Given that there are mappings that allow researchers to extract dependencies between components, it is tempting to assume designers will perform their work along these dependencies. If two components share an interface, then a change in one may lead to a change in the other or at least require that the other is aware of the change. This idea was postulated long ago (Galbraith 1973) with regards to team coordination and has been followed by others (Deming 2000b), (Eppinger 1997), (Eppinger & Salminen 2001), (Sosa, Eppinger & Rowles 2004). These assume the model of task portioning in which different designers are responsible for different components and so "if two components share design interfaces, the teams that design them need to interact" (Galbraith 1973) and, considering there are different levels of connections "the greater the interdependence between components, the greater will be the need for communication and cooperation between them." (Deming 2000b).

Based on this, one of the approaches that followed was the attempt to optimize the system to reduce the number of connections and coordination points. Some of the studies that focused on this include (von Hippel 1990), (Baldwin & Clark 1997), (Eppinger et al. 1994), (Eppinger 1997) and (Braha 2002) presenting measurement and improvement methods.

These studies laid theoretical models and methods which were used by Sosa to collect data from one specific case (Sosa, Eppinger & Rowles 2004). This study mapped the product's design interfaces and the team interactions and found that of the 569 identified design interfaces only 349 had a matching team interaction. This means that 38.7% of the theoretically required team interactions were not being completed. Another insight from this study was that there was also a relevant portion of team interactions that had no identified design interaction sustaining it. 74 events or 17.5% of interactions occurred without an apparent dependency to justify it.

There are teams that step outside of the product structure to coordinate and together they are able to apparently identify additional dependencies that need to be solved. In contrast, there is no

perspective on the individual action of project participants and how they operate with regards to the product structure. This is a gap I intend to address.
The potential of individual contributions in product development environments is immense. Taichi Ohno used the insights of individual assemblers in the production stages of product development "because of their direct acquaintance with conditions on the line" (Womack, Jones & Roos 1990) to discover defects and elicit improvements in automobile manufacturing which revolutionized that industry and many others under the "lean methodologies".
As with teams who coordinate with other teams, when designers are participating in product development efforts they have to consider the changes they are making with regards to the rest of the system. As suggested previously, they can use the dependency approach to verify the path of change propagation and make the modifications to the connected components. But as we found for teams, this method may not capture all the types of interactions occurring along the process. The following study examines this question, formulated as:

Hypothesis 2 – The elements of the product with which an engineer interacts to solve a single problem can be predicted by the dependencies in the product architecture alone.

In the next section I will describe this hypothesis in more detail and explore the experimental requirements to test it.

4.2 Research setting

4.2.1 Required characteristics and field that corresponds to these requirements

In order to test hypothesis 2 the data sources must contain sufficient detailed information regarding the involvement of engineers and the state of the product architecture. The criteria for data sources used to test the hypothesis were:

1- should represent a product development project, ideally of a complex product and spanning over several stages of the development process.

2- should allow separation of the contributions of different people and for each individual should allow individual tracking of each change, at the task level

The project setting should also guarantee minimal managerially mandated constraints so that actions by participants are not affected by them, such as:

3- engineers can decide independently and make the contribution they consider the most relevant

4- contributions can be made in any part of the product's structure, i.e., engineers are not restricted to work in a single component or sub-system of the product and can make changes affecting any components

Finally,

5- it is necessary to have detailed information on the state of product throughout it's development and

6- in each stage of the development, the structure of the product should be observable.

It was found that open source software development projects meet all the above requirements. Software products are commonly considered complex products and project development methods include the detailed recording of all modifications made to the product so it is possible to analyze past modifications. In Open Source software projects the organizational setting enables contributors to work freely on what they consider the most pertinent and to make the necessary changes accordingly. Finally, the source code for software products are digital text files and tools have been developed that allow practitioners and researchers to study the structure of the program.
In the next section I will explore in detail these characteristics of software development and conduct a review of the literature in this field.

4.2.2 Review of SW engineering domain literature

Software development has many similarities to physical product development but due to the nature of the artifact being developed, it also has its peculiarities. In this section I will describe how software can be compared to most product development efforts and then conduct a review of the academic literature that has focused specifically on software development.

What is software

Software is an integral component of many current products and systems. It consists of a set of rules to be executed by electronic components either automatically or by responding to human interaction. While software relies on and can only run with the existence of hardware, a physical platform to execute it, advances in the performance of hardware have allowed the software development to assume itself somewhat decoupled from the hardware foundation.
Current hardware platforms can execute millions of instructions per second and hold millions of bits of information in memory and storage, waiting to be processed. With this capacity, software developers are free to dream the procedures required to achieve objectives or delight users, even when they require elaborate and heavy efforts from the machines in which they are running. The availability of such computational capacity and the design of the machines in a layered or modular fashion allowed for a cognitive separation between the hardware and software. Developers of software no longer need to worry as much about the underlying hardware capabilities and are able to focus more on how to structure the operations in order to achieve the results they have as objective.
Software is a logic representation of steps to be executed and therefore intangible. It is not directly constrained by physical laws. In its development, one or more developers, write the instructions they wish to see carried out. The instructions are written in a specific syntax that can be interpreted or translated to what a machine computes and are referred to as code. Developers of software are then typically described as “coders”, “programmers”, “software engineers” or simply “developers”.
Syntaxes are also known as programming languages and programmers create software by writing the instructions in text files that then are processed by the computers. Once all instructions have been written to a file, it is ready to be executed by the computer. An example of a computer program is a routine that takes two different numbers and calculates the average between them.
This could be written in pseudo-code as:

```
Get value 1
Get value 2
Calculate average
Show result
```

Current software programs are able to accomplish much more than just calculating the average between two numbers. They can do this over collections of numbers many times larger, find information that fits certain criteria in millions of records, display information in a graphical representation, communicate and interact with other systems and process millions of instructions per second. As software systems have increased in scope so has their complexity. A modern computer operating system has thousands of tasks to attend to, each with a specific and elaborate implementation. In order to build these systems in a timely manner, many computer programmers are hired to participate in the effort.

The use of teams in software development then raises the issue of coordination and project management in the projects. Given the different nature of software when compared with physical products, the methods used in management of software projects may be different from those used in the more conventional product development projects. In order to better understand the peculiarities of software development, I reviewed academic literature that has studied this field.

Sampling the literature of software engineering

In order to guarantee a sufficient understanding of software development processes I studied two technical books on the subject (Sommerville 2006; Tomayko & Hazzan 2004) and sampled articles that addressed some of the different issues pertinent to this thesis, such as coordination of teams, project management and product architecture. After collecting and reviewing a set of 50 different papers, I also analyzed their list of citations in order to find, from the whole set, if there had been a common important reference that the original sample had missed.

The Software development process compared to classic product development

As noted above, software is constrained by logic and not by physical constraints. Classic product development literature typically focuses on electro-mechanical devices such as automobiles, printers and roller skates. The generic development process for these products follows four steps:

1 - Conceptualization,

2 - Design,

3 - Optimization,

4 - Validation.

Pahl and Beitz, in their European reference book (Pahl & Beitz 1996), have a slightly different take on the different steps constituting the product development process with:

1 - Product planning and clarification of the task,

2 - Conceptual design, and

3 - Embodiment design.

Another, more refined description of the steps can be found in Ulrich and Eppinger (Ulrich & Eppinger 2000) which was already described in Figure 1. In this book, the stages are:

1 - Planning,

2 - Concept development,

3 - System-level design,

4 - Detail design,

5 - Testing and refinement,

6 – Production ramp-up.

Software products are, by nature, different from those analyzed in these two references, but the development process for these is very similar also according to a book that is a reference in industry and academia. In "Sofware Engineering" (Sommerville 2006) the author offers three different development models for software. In the first, the "Waterfall" model, the development process takes five steps:

1 – Requirements analysis and definition

2 – System and software design

3 – Implementation and unit testing

4 – Integration and system testing

5 – Operation and maintenance

By comparing the electro-mechanical development processes and the software processes it is easy to observe that the high level descriptions are very similar. Each model describes a maturation of an idea into a system which then is decomposed into different units for development and subsequent testing.

The other two models described by Sommerville are the "Evolutionary development" and the "Component-based software engineering". The premise of "Evolutionary development" is that new information is generated by the development and usage of the product so it can be continuously refined. It is more effective at tailoring and meeting the user's requirements sooner and more narrowly. As a drawback, it is a less visible process for managers. In comparison with the more classic product development literature, this is very similar to the concept of "spiral development" which also first originated in the software development world (Boehm 1986).

Finally, "Component-based software engineering" advocates the reuse of portions of code from other projects and the use of an integration framework to connect the different portions. Several studies in the classic product development literature have also addressed this issue under the terms of modularity and commonality in systems.

Essentially, both approaches analyze how different blocks and functional units are able to be designed and function almost independently and then integrated into a system that conveys the entire required functionality. One of the advantages of this approach is that, with well established interfaces between modules, the inner workings of each are irrelevant to the rest of the system so they can be worked on without requiring or propagating changes to other components.
With this comparison, we can see how similarly software developers and electro-mechanical product developers think of their processes. Despite the difference in nature, the way the processes and the product architectures are conceived are very much alike, which lends software to be a subject of study even for the interests of classic product development researchers and practitioners.

Two different approaches to software development: Open and Closed Source

Currently, there are two main tracks of software development: closed-source and open-source. Closed source projects typically belong to a single organization responsible for its development and is called "closed source" because the code that is developed is maintained hidden from the users. Many commercial software projects are of this nature and their development teams are co-located and the waterfall approach can be used.
The other track is open-source software. In this type, the code that is written for the software product is publicly shared and available. This allows anyone to study how the program is working and to make changes that they need for their objective. Changes that are made can be sent to the managing team for inclusion in the main product and made available to all other users. This practice is supported by a legal contract which states that a user is free to modify the code of the software for his objective and, if distributed, that the resulting code has to be openly shared again (Free Software Foundation 2007).
Closed source software development was the norm for most projects until the 1990s. In that decade a set of open source projects started to emerge and gain public visibility. Open source projects introduced new ideas not only in the development of technology projects but also to the licensing of intellectual property and have been the subject of many researchers who believe or have found some evidence that its approaches are relevant to other areas of economic and social activity (von Krogh & von Hippel 2006).
Since all the code and development information is publicly accessible there is a research opportunity which is rich for data collection. The two main advantages of data collection in this type of projects are:

1- Open source projects and the way they are executed generate a lot of descriptive data. A researcher is able to collect and study this data without interfering with the project execution or distracting its participants.

2- The record of descriptive data is typically kept for the entire lifetime of the project so participants can trace the root cause of an issue. This is also advantageous for researchers by allowing them to study the long development history instead of a brief period in the project's lifecycle.

Other advantages include the level of detail of the information (very fine grained), consistency of data formats over time and the quantity of information generated, even by small projects.(Mockus, Weiss & Zhang 2003)

Individual roles in software development

A programmer in a software project writes pieces of code that describe the tasks to be performed. As the program grows in complexity, different files store different parts of the code. When working collaboratively, a programmer grabs a portion of the code to work on and, once done, shares it again with all other programmers. As the different functionalities get written, they are tested against a set of parameters to ensure it is working properly. Sometimes a piece of code that seems to be working produces an error which is known as a bug. Programmers then have to trace the source of the error in order to fix it (known in the industry as a "patch"). Fixing a bug follows a process that tries to document the different steps taken between when it is first observed, reproduced, fixed, the fix is tested and the bug is considered closed (The Bugzilla Team 2010).

Coordination in Software projects

Teams involved in software development also need to be able to solve tradeoffs and exchange information in order to guarantee that the different portions and tasks they are dedicated to will work well once integrated. Open-source projects mostly rely on the contribution of people scattered around the world who contribute opportunistically. With such a large geographical dispersion of members, face to face meetings are not possible and the projects have to use communication technology like the internet to share information and coordinate all the contributing participants. One of the previous studies of open source software analyzed how software development teams of open-source projects coordinate their actions (Cubranic & Booth 1999). In this research they found that teams employed low-level tools such as email, mailing lists and bug trackers to exchange information and make decisions. Even with the use of simple tools, different types of decision making organizations emerged. Some projects have a single decision maker who approves what features shall be included or not in each cycle of the product (for example, the Linux project) while others rely on a committee vote to decide such (for example, Apache). Finally, there are systems that are decomposed into a set of independent subsystems and, in these, participants make their own decisions (for example, Mozilla).

Since several open source projects have produced strong products, some researchers have suggested and looked into the applicability of this methodology to closed source and additional different types of efforts.(Mockus & Herbsleb 2002)

An essential tool used in almost all serious software development projects, regardless of scale, is the use of code repositories (Halloran & Scherlis 2002). A code repository is a software program itself that is responsible for archiving of all the software code files of the product being developed and a history of changes made over time to the code. These repositories serve two main purposes: allow simultaneous collaboration and allow reverting the code to a known working condition.

For the first purpose, when multiple people are working collaboratively, they share a common pool of code which they edit together. The repository has the responsibility of checking if people are working on independent parts of the project and, when there is overlap or conflict, it alerts the user so a choice can be made. For example, an extremely simple project has just 2 files of code. If programmer A works on one file and programmer B works on the other file, the system will allow them to make and submit changes simultaneously. If they try to make a change to the same file, one of two things can happen, depending on the sophistication of the repository. The repository may block the second user from making changes while the first user isn't finished or may conditionally allow changes to be made. When the two changes are saved, an analysis of the code is done and if the changes, although within the same file, are in different sections of the code, then the changes are merged together.

The second purpose of a repository is maintaining a history of changes. If at one point the programmers find that a previously functioning feature is broken, they can go back (revert) to a previous known state, and undo all the subsequent changes.

The use of this tool assists the teams in guaranteeing a common, centralized version of the software product which synchronizes all the team members on the latest developments. This is an example of a method that centralizes information that was described in chapter 1.

Since each programmer copies the latest common version available to work on, it is required that when changes are made and tested, they are readily made available to all other contributors so they can keep working on the latest versions and avoid working with outdated states of the product.

Also, it is advised that each individual change be submitted to the repository independently from other changes. A programmer could work on multiple tasks and then submit the finalized collection to the common repository. While possible, the development communities avoid this practice so that, in case a reversal is required due to a problem encountered later in the development process, only the function causing the problem is reversed. It also allows for easier inspection of modifications when a change is submitted.

Social settings, motivations and behaviors of participants

Due to its decentralized, open nature, the motivations and behavior of participants in open source projects is also peculiar, although in such a way that is relevant to the objectives of this study. Individuals participate in these projects paid or voluntarily and are motivated mainly by reputation and recognition, improvement of one's own software and career advancement (Hertel, Niedner & Herrmann 2003). It has also been suggested that they also participate due to social and political ties or for the fun of programming (Lakhani & Wolf 2003) and that all these factors are interrelated (Roberts, Hann & Slaughter 2006). Another characteristic of open source projects is that work is not assigned to any individual. Contributors are free to work and contribute in their areas of interest and select to work on the pressing problems that they are skilled for (Mockus & Herbsleb 2002). However, there is some variation on what type of work is done based on experience and longevity with the project, typically with novices selecting to work just in the areas in which they are personally interested and then evolving into other areas, even the unattractive and seemingly mundane and repetitive but necessary tasks of community organization and housekeeping (Bagozzi & Dholakia 2006).

Social Network analysis

As the complex software development projects require teams of programmers working collaboratively, this is an environment that lends itself to also be viewed from a social network analysis (SNA) perspective. A multitude of SNA research has been conducted in the recent years focusing on many different subject communities.

Research that used SNA and focused on open-source communities found that the structures of the teams can vary, with some having a programmer in a highly visible central node coordinating all the work and other with a decentralized communication path between programmers (Crowston & Howison 2005). This contrast correlated with the size of the project, with much larger projects having a more decentralized structure.

Another view looked at the association between programmers over 39 000 projects and found evidence suggesting these communities have self-organizing properties. It was also observed that in the larger projects there is mostly no central control or planning (Madey, Freeh & Tynan 2002).

Measures of software structure and complexity

The ability to measure the structure or complexity of a product has always been of interest to productivity researchers. There are several metrics used in the physical space, like part count, that are hard to apply in the software world. As any, the metrics used in the software domain also have raised disputes regarding their accuracy and different generations of research have favored or ignored different sets of metrics. In another case, some metrics are used because they are very simple to calculate and consistent across projects. Such example in software is the use of lines of code (LOC).

The lines of code metric consists in simply counting how many lines a program has. A line of code roughly represents one instruction so many lines of code should represent a more complex product. This is not always a correct assumption since programmers use loops in the code, running some lines more than once or reusing portions of code for different parts. Take for example a code that calculates and prints the square of numbers between 1 and 9. Two distinct ways to write it could be:

First example:

```
Print 1*1
Print 2*2
Print 3*3
Print 4*4
Print 5*5
Print 6*6
Print 7*7
Print 8*8
Print 9*9
```

Second example:

```
For i in range (1, 9):
        Print i^2
```

The first approach produces correct results but is very bad form for software programmers. The second is how such should be written since it reduces the repetitive instruction but when analyzed under LOC metric, the first approach seems more complex, while they are both doing the exact same thing.

A more modern approach consists of using the Design Structure Matrix (DSM), informed by the Function Call Graph (FCG). The DSM is a representation of dependencies in a square matrix. It is a method extensively used by researchers in the context of product development to map dependencies such as physical, energy and information connections, as well as process (tasks that depend on other tasks) and social dependencies (who talks to whom). The DSM representation can immediately be converted to and from a network graph, although many of the manipulation techniques rely on a matrix representation for data extraction, such as the identification of modules.

The Function Call Graph is a technique for extracting dependencies in software so they can be represented and analyzed in a DSM. Often times, programmers write blocks of code that perform a specific, limited function which then is accessed (called) whenever it is necessary. By analyzing the code it is possible to identify what parts of the code call other parts, typically at the file level.

Using this approach, one which I will also rely on later in this thesis, different studies have analyzed software projects.

One compared the outcomes of open and closed source software projects (MacCormack, Rusnak & Baldwin 2006) and found evidence supporting theories regarding that open source projects can create more modular architectures than closed source projects.
Conway's law is a design theory which states that the structure of a designed product will resemble the structure of the organization which built it (Conway 1968). This is also known as the mirroring hypothesis and a test of this hypothesis was done using similar open and closed source projects and comparing the paired outcomes (MacCormack, Rusnak & Baldwin 2008).

System evolution

With the ability to analyze a frozen state of product, researchers were able, as long as detailed data was available, to extrapolate the same type of analysis to more than one instance along the development path. This allows us to look at the temporal domain and into how systems evolved. Studies which have done this include (MacCormack, Rusnak & Baldwin 2007), (LaMantia et al. 2007), both of which have used the DSM technique.
Other approaches include the adapted use of the rudimentary count of lines of code method (Godfrey & Tu 2000), graphical visualization techniques (Lanza 2001), rate of change in simple project and lines of code metrics (Koch 2005) and relating the system being developed with the community sustaining the effort (Nakakoji et al. 2002)

Selected case studies

Different case studies have been published that follow in detail a project or a set of open source projects. They have focused on:

- server software such as the Apache project (Mockus, Fielding & Herbsleb 2000) which has been one of the most widely used web servers in the last decade and a half,
- the GNOME (Koch & Schneider 2002) and KDE (Hemetsberger & Reinhardt 2004), the two most popular desktop environment packages for consumer operating systems.
- the Mozilla project in its transition from Netscape, a closed source operation, to an open source development (Mockus, Fielding & Herbsleb 2002)
- FreeNet, a decentralized network infrastructure (von Krogh, Spaeth & Lakhani 2003)

Open source projects rely heavily on visibility to be able to attract new collaborators. The previous four projects are big enough and well known that they can attract participants. For smaller, niche projects or development projects simply starting, several hubs that aggregate projects and developers have been created to host these smaller projects. They provide most of the necessary tools to run a distributed open-source project. One of the most popular is SourceForge and it has also attracted

researchers who, more than study individual projects, wish to compare between several projects in order to test their hypothesis.
One example of work conducted on hub services like SourceForge are a survey of 100 projects (Krishnamurthy 2002) which found that not all open-source projects are developed by a community and that, in his sample, the majority were actually lone or almost individual efforts.
Another paper that analyzed community building, mined 7477 projects (Crowston & Scozzi 2002). This experience informed another paper by one of the authors which made an analysis of the merits of mining these collective repositories (Howison & Crowston 2004) but also determined that a mere analysis at the project metrics level ("Number of developers, Project status, Activity, Downloads, Page Views") can lead to poor hypothesis testing and conclusions. The authors recommend a deeper dive into each project to fully understand what happened.

Summary

In this section and through a review of academic literature I showed how software development is similar and relevant to the generic study of product development.
I also characterized how participants in open source software development contribute and are free to work on the parts of the project they are most interested in and share a common, accessible repository.
I described how it is possible to observe a project task by task and finally, I highlighted some of the techniques used to study software structure, complexity and evolution.

In the next sections, all these aspects will be combined in the research approach I followed to test my hypothesis.

4.3 Data Collection and Experiment setup

In addition to the criteria established in section 4.2.1, the projects selected to be studied should also have been in development for a considerable amount of time and their product developed enough to be in active use for its targeted purpose. The projects selected were Chandler, Zope, Twisted and Trac which have easily accessible development history information.. The objective of each of these projects is different from one another (a personal information management application, a web server, a framework for networked applications and a project management web application) so that results are not biased by the type of product being developed.
As will be seen later in this document, the analysis of each project is extensive and time-consuming, even when automated, so only four projects were considered for the sample.

Chandler

Chandler is a personal information management application. The development of Chandler was led for a long time by Mitch Kapor, an experienced and notable software programmer who, among other achievements, founded Lotus Development Corporation and designed Lotus 1-2-3, one of the earliest and most popular spreadsheet programs. Kapor (Kapor 2010) set up an organization named OSAF - Open Source Applications Foundation through which he funded the development of Chandler.
The product's objective is to enable end-users to manage their agenda, electronic messages and notes and their inter-relationships in one integrated application. Its development started with a vision contrasting with the then standard process that users followed and a desire to replace the incumbent products. The vision and the reputation of Mitch Kapor gave the project sufficient attention and was considered by some of the leading universities and enterprises as a promising product. Its development was followed closely by a journalist who documented, in an ethnographic piece, the first years of the projects development (Rosenberg 2008).
The work on Chandler started in August 2002 and this study followed its development until November 2009.

Zope

Zope is a sophisticated web application server featuring an integrated database engine to store the application information. It is one of the leading applications using the Python programming language and is currently used by large enterprises such as GE, Viacom, Verizon Wireless, NASA and the US Navy to run their websites (Zope.org 2010)

The available data archives on Zope development go back to 1996 and this study followed its development until September 2009.

Twisted

Twisted is a networking framework that supports multiple internet protocols. It is a product targeted to other programmers that may use it as a networking abstraction layer in their applications. By using Twisted, programmers can reuse the networking implementation that the product offers and focus on the logic of their application and not spend time coding network messages or following the communication protocols.
The data archives containing the development of Twisted start in 2001 and this study followed its development until September 2009.

Trac

Trac is a project management tool, primarily used in management of software development projects. It offers typical project management features such as roadmaps and milestones as well as a tickets system that allows managers to follow tasks and/or bug solving work as they are completed. Another feature offered by Trac is a wiki engine which allows project participants to collectively create and modify shared documents. Trac is also able to integrate with some software development repositories, automating some tasks for software developers although it can be used for projects which do not include software development. Among the many organizations that use Trac are General Dynamics which uses it "for development and documentation of spacecraft structural analysis projects" and NASA's Jet Propulsion Laboratory which uses it to "manage various deep-space and near-space projects"(Trac Project 2010). Another user of Trac is Twisted, one of the projects included in this study.
The data archives containing the development of Trac start in 2003 and this study followed its development until June 2009.

4.3.1 Project data

For each of the projects a copy of their code repository and changelog was downloaded.

Project Repositories and local copies

A local repository was created from the downloaded files, replicating the development environment seen by the project's programmers up until the day when the download was executed. The repository was downloaded using the tool "svnsync" (Apache Subversion 2010), which is employed for backups. With the local copy it is possible to extract the source code of the project in any of its steps.

The changelog was extracted using TortoiseSVN (Tigris 2010) and saved into a text file. This file includes descriptive information regarding each change made in the product, namely the author, date, message and a list of the files changed, created or deleted. All or most of the developers in these projects have commit authority. There is no visibility over which developer made the contribution if he does not have commit authority. Since this study does not focus on the different profiles of individuals making contributions, but on the types of contributions made by all, it is irrelevant in this case of who had the commit authority. For this study what matters is that the change exists and is being incorporated into the product.

An example of one such change in the Chandler project:

```
Revision: 7399
Author: pbossut
Date: 8:22:00 PM, Thursday, September 22, 2005
Message:
Fix bug #3390: suppress separator in the markup toolbar
----
Modified : /trunk/chandler/parcels/osaf/framework/blocks/detail/detailblocks.py
```

Figure 4 - Project change log sample

The datasets for each project span different durations and include a different number of changes. The magnitude of changes indicates that all four projects are similar in size and scope. The following table summarizes these values

Project	Time start	Time end	Number of changes
Chandler	August 26, 2002	November 18, 2008	14835
Zope	June 17, 1996	September 3, 2009	10974
Twisted	July 8, 2001	September 6, 2009	15289
Trac	August 10, 2003	June 2, 2009	8263

Table 9- Dataset of Software Projects

Analysis testbed

With a local copy of project data, the next step was to create the conditions for analysis. A local database was set up and the information in the changelog was extracted into two tables per project. One table contains the descriptive information of each change (revision number, author, date, development days and message) and the other the list of files changed per revision.

Python is a popular programming language which is also the language used in the four projects analyzed and was also selected to write the analysis tools. The python software programs, also known as scripts, were used for all the different steps of the analysis, starting with the parsing of the changelog text file and inserting the information in the database.
The python programming language also offers a reasonable set of library extensions that allowed performing analysis without requiring writing specific implementation code, such as for network analysis algorithms or interfacing with the database for exchange of information.
The different python libraries used were:

MySQLdb – interfacing between python scripts and a MySQL database
NetworkX – offers a library of graph management functions and algorithms
PySVN – controls a subversion repository from a script
Heatmap – creates a heatmap representation from a large set of scatter data
Numpy – scientific computing library
Matplotlib – plotting library
Python Imaging Library – image processing library

Some of the analysis steps, even automated in software, required extensive computing times so more than one computer was used. These steps required more than one week of continuous operation using current desktop consumer computers.
The different algorithms will be described in more detail in section 4.4

4.3.2 Function call graph

In order to test the hypothesis, it is necessary to obtain a representation of the structure of the product. In software, one of the possible approaches to do this is through the use of the function call graph.
The function call graph is a network representation of all the call dependencies between the different files that constitute a program. The dependencies result from pieces of code including a request (call) for a function already written and available in a different file. This promotes code reuse and sharing, allowing programmers to focus on writing the feature they are aiming for without having to write procedures already previously written. For example, mathematical functions may be written in a separate file which is called when a programmer needs to use, for instance, a square-root operation on a number variable.

The scripts written for the analysis of these datasets use this feature abundantly with the libraries mentioned in the previous section. By calling these libraries, I was able to focus on the test algorithm and not on implementing the communication protocol with a SQL database, for example. By mapping all the calls between all the source code files it is possible to extract the graph of a software program which can then be analyzed. This method is used both by practitioners and researchers of software alike. Practitioners use it to search for potential security vulnerabilities (Wagner & Dean 2001) or to improve global performance (Laplante 2000).

Researchers have also devoted time over the years on how they can be built (Ryder 1979; Murphy et al. 1998; Grove & Chambers 2001; Callahan et al. 1990; Allen 1975). Researchers have also used the call graph as a tool to understand the evolution of software structure (Collberg et al. 2003) as well as a comparative basis between different models of organizational setup (MacCormack, Rusnak & Baldwin 2006).

Call graphs can be of two types: static or dynamic. Static call graphs are the most comprehensive as they map every possible dependency in the program. Dynamic call graphs are those which record which calls are made during a particular or a set of executions of a program. The dynamic call graph addresses the fact that it is not necessary that all available functionalities of a program are used when a program is run.

This is analogous to say that the horn in an automobile, while a feature that is available, is not used every time someone drives a car.

Since I am most concerned on how the designers perceive and interact with the system they are building, I will focus on the static call graph. Designers take into account all the possible dependencies in the program and rarely focus on a single instance of execution that can ignore a specific call. This also facilitates the extraction of the call graph as it can be directly obtained by the analysis of the source code.

The four projects analyzed here are python programs so the same extractor can be used for the four projects. There are some programs that extract this graph given a set of source code files but none of them were usable for this project. As these tools are used to analyze a single instance of a program, they did not scale to the objective of extracting and analyzing the thousands of versions I aimed to look at. This required me to write my own extractor which took a revision of the projects being analyzed (a step in their development process) and analyzed all the files in it and extracted the calls originating from each file.

Some considerations:

- only files with software code were analyzed. I ignored any files with documentation, images or graphics.
- I matched every line of active code against the Python documentation specification for how to make a function call. From Python version 2.5.2 documentation the call syntax is as follows:

```
import_stmt  ::=  "import" module ["as" name] ( "," module ["as" name] )*
                | "from" relative_module "import" identifier ["as" name]
                  ( "," identifier ["as" name] )*
                | "from" relative_module "import" "(" identifier ["as" name]
                  ( "," identifier ["as" name] )* [","] ")"
                | "from" module "import" "*"
```

So far, this is how most source code call graph extractors operate. My code also implemented a comparison between different revisions. Once all the calls from a specific revision number had been extracted they were compared with the set of calls from the previous revision. The difference between the two groups revealed which calls were common and already existed, which were new and introduced in this revision and which were no longer present and had been removed.

All the changes in a project were registered in a text file which included the revision number in which the change was found, the type of change observed (added or removed), the file making the call and the target file. This is the first set of calls in the Chandler program:

```
9       ADD     chandler.chandler      cal.calendarview
11      ADD     chandler.application.chandlerwindow   chandler.application.menubar
11      ADD     chandler.application.chandlerwindow   chandler.application.locationbar
11      ADD     chandler.application.chandlerwindow   chandler.application.actionsbar
11      ADD     chandler.application.chandlerwindow   chandler.application.sidebar
11      ADD     chandler.application.sidebar  chandler.application.navpanel
12      ADD     chandler.chandler      chandler.application.chandlerwindow
12      REM     chandler.chandler      cal.calendarview
...
```

From this data it is possible to reconstruct the function call graph of the project at any revision number. To do this, one just needs to add and remove the edges of the graph from the start of the project up to and including the changes of the desired revision number. Some further notes on how data was registered:

- I am connecting files to files, so instances in which only a specific function is called from a file is considered as similar to a call to all functions in a file

- I also matched conditional calls, i.e. calls that are only made depending on the conditions in which the software is run (static call graph)

Finally, calls that are made to generic external libraries, i.e. parts of the code used by the program but not developed for it, were removed from the analysis. Python, the language in which all projects were developed, offers basic programming constructs and functions out of the box but has many libraries available, such as those that interpret and parse time values, advanced math operations and operating system interactions.

4.3.3 Network analysis for system analysis

The analysis of socio-technical systems represented in the form of graphs has provided many methods and applications (Wasserman & Faust 1994) and is still a very vivid research field (Borgatti et al. 2009).

The nature of software architecture and specifically in the point of view of the function call graph lends itself to be studied with the use of these methods and metrics. The analysis of the structural network of a software program will enable the researcher to find system wide dependencies, identify sub-modules and characterize different components as more important than others to operation of the program.

In network analysis, centrality measures are used to determine the relative importance of a node in a network and the existing connecting paths between nodes. One of the popular measures of centrality is betweeness centrality which counts how many of the shortest paths connecting every pair of nodes in a network pass through a given node. To calculate the betweeness centrality measure of a node we first must calculate all the shortest paths between all the pairs of nodes which, given a large network may be computationally demanding, and from those paths count how many times the given node is present.

An alternative centrality measure focuses on the local neighborhood of a node, such as degree centrality, which measures how many nodes have connections to it. Betweeness centrality is a more adequate measure of software systems mapped using the function call graph since the calls are often transitive in their nature. If a portion of code A makes a call to other portion of code B, when A is called by another piece of code it will also rely on calling the B portion. The dependencies are inherited and so it makes sense to evaluate the entire path of calls, rather than just the direct neighboring nodes.

The figure below represents a simple call structure. In this example, node B is the single node that is common to all the nodes. By removing or damaging the operation of this node all functions may be compromised whereas by removing node A would only compromise the operation of a few nodes.

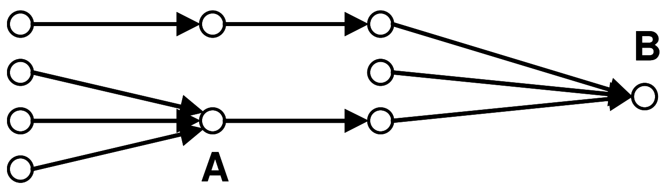


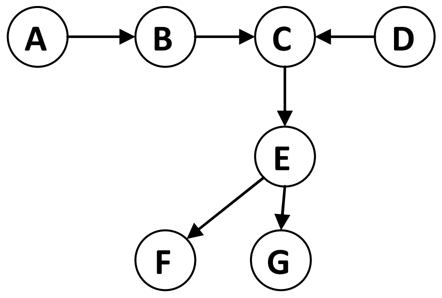
Figure 5 - Simple network structure

4.3.4 Structural correspondence

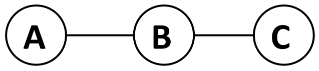
When programmers attempt to make an improvement to the program they change or add code and may do this within a single file or over the span of several new or existing files. As is described in the review of software development literature, the rule of conduct for open source communities requests that as soon as a task has been completed it is shared with the other programmers by updating the common repository and giving everyone access to the latest copy. The data describing each task solved is available and can be immediately retrieved from the database containing the changelog.

In the hypothesis being tested, this information constitutes the "elements of the product with which an engineer interacts to solve a single problem". The second part of the hypothesis states: "… can not be predicted by the dependencies in the product architecture alone". This is the information we can extract from the function call graph.

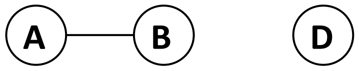
In order to compare the changes made with the structure of the product I build two networks, one resulting from the revision information and the other from the function call graph, selecting the nodes present in the change. I then analyze the correspondence between the two and evaluate if the files changed are represented by a connected graph in the function call domain. For example, at one point of its development a small program has the following function call structure:



Scenario 1: A programmer changes files A, B, C. From the call graph we can observe that these files are all connected, and so the programmer's change corresponds with the structure of the program.

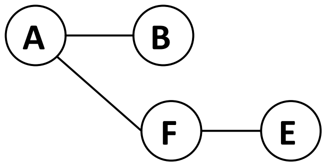


Scenario 2: A programmer changes files A, B, D. From the call graph we can observe that these are not directly connected and does not correspond with the structure of the program.



Scenario 3: If there is a single change to file A, it is only possible to represent the node and we can not compare it to the function call graph. Changes to single files are most likely internal improvements to that file and will be ignored in the testing of the hypothesis.

Scenario 4: A programmer changes files A, B, E, F and during the change introduces a new call between A and F. The call graph after the change is made is the one considered when comparing the two groups as the programmer, when making the change, had in mind creating the dependency, therefore what he worked on corresponded to the final structure.



These different scenarios explore the correspondence between the files that are changed and the structure that the software product has. This comparison will be the base of testing hypothesis 2.

4.4 Data analysis & Results

4.4.1 Hypothesis 2

This hypothesis states that: "The elements of the product with which an engineer interacts to solve a single problem can be predicted by the dependencies in the product architecture alone."

From the data collection and testbed setup I have shown how to observe what components are changed and how to extract the dependencies of the product at any given point in its development. For a given revision being analyzed, the files that are changed by a programmer are obtained from the changelog database. If only one file is changed in that revision we ignore it and skip to the next revision number.

When more than one file is found, the function call graph of the project right before the change was made is built and then compared with the call graph of the revision being tested. By a simple comparison of links it is possible to determine if this change added or removed links.

Finally, from the call graph representing the revision number being tested I select only the sub-graph containing the files changed in this revision and evaluate if they form a connected graph. This indicates if there is structural correspondence between what the programmer changed and the structure of the product. The result is registered in a results file for later processing and the steps are repeated for the next revision until an entire project is analyzed.

The vast majority of changes affect only a single file. The numbers for the four projects are:

Project	Total number of revisions	Revisions with changes to a single file	Percentage of revisions with changes to a single file
Chandler	14835	10887	74%
Trac	8263	7005	85%
Twisted	15289	11884	77%
Zope	10974	8755	80%

Table 10 - Software project statistics

These are mostly local changes which can be dealt with without considering the system wide implications. I am more interested in the changes that have wide impact and require that more than one file or component be changed at the same time. These are the changes that require that the programmer consider the connectedness of all the files being changed.

The number of code files that are changed in conjunction with other files also varies, as summarized in the following table.

	Project			
Number of code files changed	Chandler	Trac	Twisted	Zope
2	39%	43%	54%	58%
3	19%	17%	17%	14%
4	11%	9%	8%	9%
5	8%	6%	5%	4%
6	5%	3%	3%	3%
7	3%	3%	2%	2%
8	2%	2%	2%	2%
9	2%	1%	1%	1%
10	2%	1%	1%	1%
11	1%	1%	1%	0%
12	1%	1%	1%	1%
13	1%	1%	0%	0%
14	1%	1%	0%	1%
15+	6%	10%	3%	4%

Table 11 – Frequency of changes to sets of multiple files

Also, while most files are connected through calls or dependencies to other files, there are some that can exist on their own, disconnected from the rest. For the four tested projects the number of files that exist independently and disconnected from other files in the projects is very low. The files that are not connected to other files in the program are called singletons in this document. When couting singletons, files that are in the "sandbox" (an area reserved for developers to experiment solutions and which is not part of the main program) and files moved to folders with deprecated code are not counted.

Project	Number of singletons / nodes
Chandler	27 / 624
Trac	17 / 269
Twisted	26 / 854
Zope	70 / 1419

Table 12 - Number of singleton files

When trying to understand team coordination we are not so much concerned about what the teams achieve independently within their subdomain and the focus goes towards the interactions that happen across the different teams. This situation is analogous to when only a single or multiple files are changed.

Analyzing the revisions that change more than one file allows us to compare their connectedness from the engineer's perspective and the product's structure. For a given task, programmers select a set of files to change based on the task objectives. There is an implicit argument why those files are part of the same change. In this hypothesis it will be tested if the reason they are associated together is due to a propagation or association between the files related to how they are connected in the product structure.

This follows from the postulate that "if two components share design interfaces, the teams that design them need to interact" (Galbraith 1973) but at the individual change level. Does the engineer also follow the structure of connectedness between components?

In the software domain I have shown how the function call graph can create a representation of the structure of the product. Now it remains to test if the set of files that are changed as part of the same revision are all connected in the function call graph.

As detailed in the previous section, this comparison is made for all revisions through the project's development. Once all revisions have been evaluated an aggregate measure is taken revealing the following numbers:

Project	Revisions with changes to more than one file	Revisions with structural correspondence	Percentage of changes with structural correspondence
Chandler	3948	1042	26.4%
Trac	1258	293	23.3%
Twisted	3405	479	14.1%
Zope	2219	365	20.8%

Table 13 - Structural correspondence in software projects

The results obtained revealed that there is some structural correspondence, i.e. all the files that are changed together are also connected in the functional call graph but the numbers fall well below what was expected from the extrapolation of Galbraith to individual actions. The structural correspondence value given by a random association of files being jointly changed is under 0.3%. These values are consistent in their range across the four projects analyzed (20% ± 6%) so the case of being in the presence of an outlier is not strong. A sensitivity test allowed for a percentage of links to be missing and still have the change considered structurally correspondent. For example, a change to 20 files in which only one of those files is not connected to the others will not be considered a structurally correspondent change in Table 13, but it could be argued that it is only a small percentage of the change that is not connected. To test the sensibility to these cases the analysis for the four projects was redone and a percentage of missing links was established as the threshold to be considered structurally correspondent. This threshold was varied between 0% (all files must be connected to be considered a structurally correspondent change) and 50% (there are as many links missing to connect files as there are non-redundant links connecting). This variations was tested in 5% increments and the variation in percentage of structurally correspondent changes is illustrated in Figure 6.

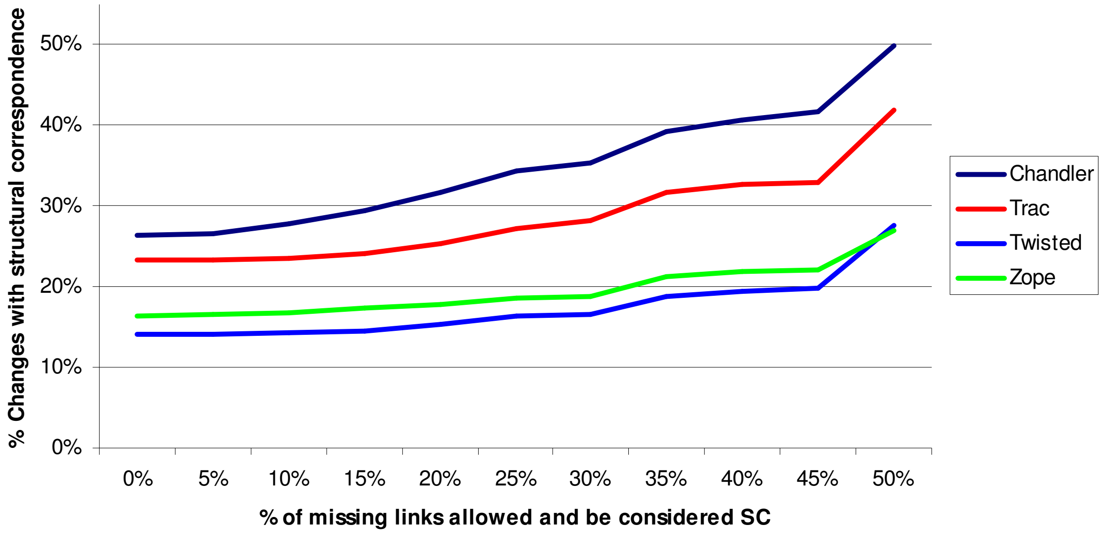


Figure 6 - Sensitivity test to absence of a percentage of links in structural correspondence threshold

The raise in percentage of changes considered having structural correspondence is shallow, revealing that the use of a 0% threshold to characterize structural correspondence is not threatened by sensitivity to few links missing from the change call graph correspondence.

Previous studies that had focused on the correspondence between structure and team interaction found the opposite results. For two studies, one of which was dedicated to a software development effort, 85% and 82.5% of team interactions matched the structure of the product (Sosa, Eppinger & Rowles 2004; Sosa 2008). The team interactions that had no corresponding product structure were the rare occurrences.

By contrast, the studies in this thesis focused on individual interactions and associations of different product components, not team interactions, and the results are the opposite. Most changes do not

follow the defined or resulting structure of the product. Pieces of the product are being changed jointly without a dependency or link to support their grouping.
While the results do not support the hypothesis they in turn raise additional questions. One is that since the calculations are the percentage taken over an extensive time of project development, the results may not reflect the evolving nature of the product. Another reflects on the idea that the work being done in the project is focused on extending the product and, from a network perspective, may lie closer to the periphery of the graph, in a less stabilized portion and so with less chance for connections to observe structural correspondence.

These two ideas will be further explored as additional hypotheses in the following sections of this document.

4.4.2 Hypothesis 3

Galbraith's reasoning and its extension to individual action presupposes that the dependencies exist and that the system has stabilized its links such that the different actors may follow them. In new product development there are several stages in which the uncertainty of the product design may preclude project participants from knowing what the product structure actually looks like and this will only emerge after several iterative development steps.
Since the data analyzed in hypothesis 2 encompasses work done in all the stages, it is possible that the aggregate correspondence between what people work on and the structure of the product is affected by low correspondence in the early stages and higher correspondence in the stages where the product has matured.
Following this line of thought, I establish the following hypothesis:

Hypothesis 3 – The amount of work without correspondence to the product structure diminishes as the design evolves.

This hypothesis can be tested by simply adding a temporal detail to the data analyzed in hypothesis 2. This data is directly available from the changelog and is converted to the number of days in the development of the project. When plotted in intervals of 120 days, project Chandler reveals:

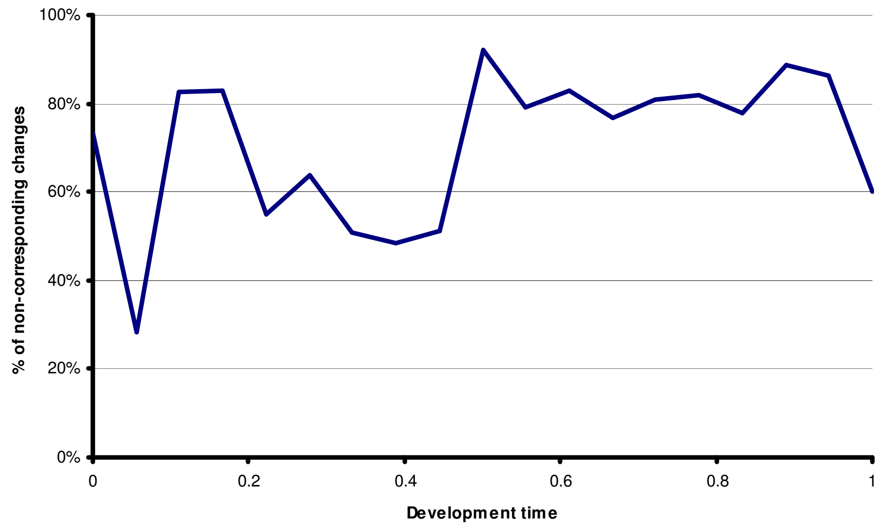


Figure 7 - Evolution of non-corresponding changes in Chandler

This figure shows evolution of the percentage of the revisions which change multiple files that do not find a correspondence between being changed together and the function call graph. What this chart shows is that there is no decrease in the amount of revisions without structural correspondence and that there is also no increase in the amount of changes where it is possible to find a correspondence between the selected files and the product structure, thus not supporting the hypothesis.

For the other three projects the charts are as follows:

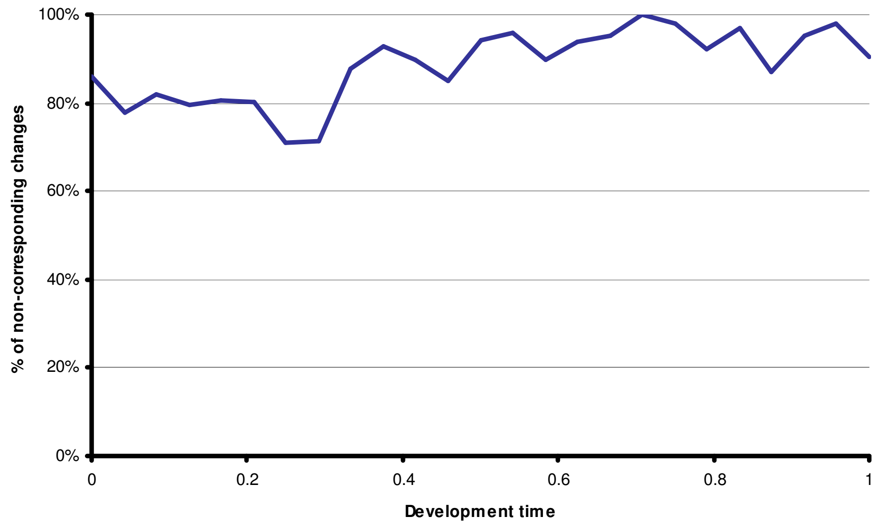


Figure 8 - Evolution of non-corresponding changes in Twisted

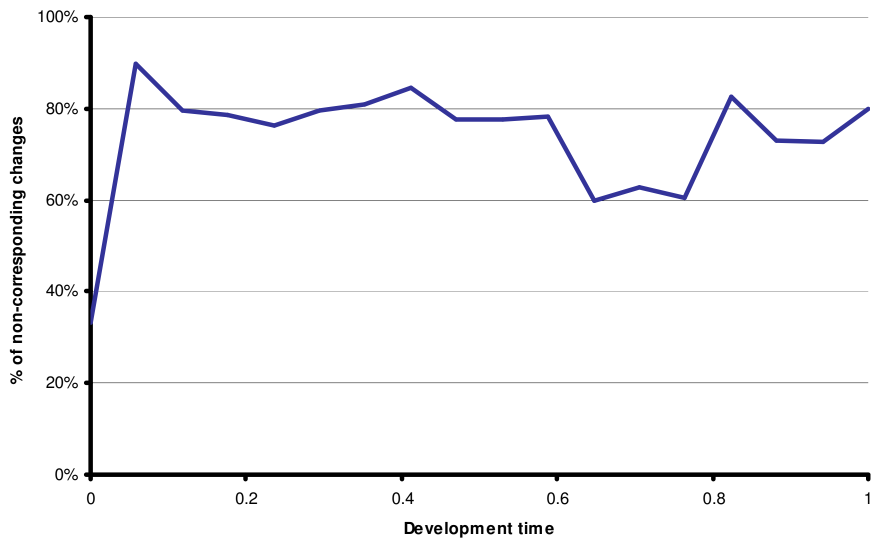


Figure 9 - Evolution of non-corresponding changes in Trac

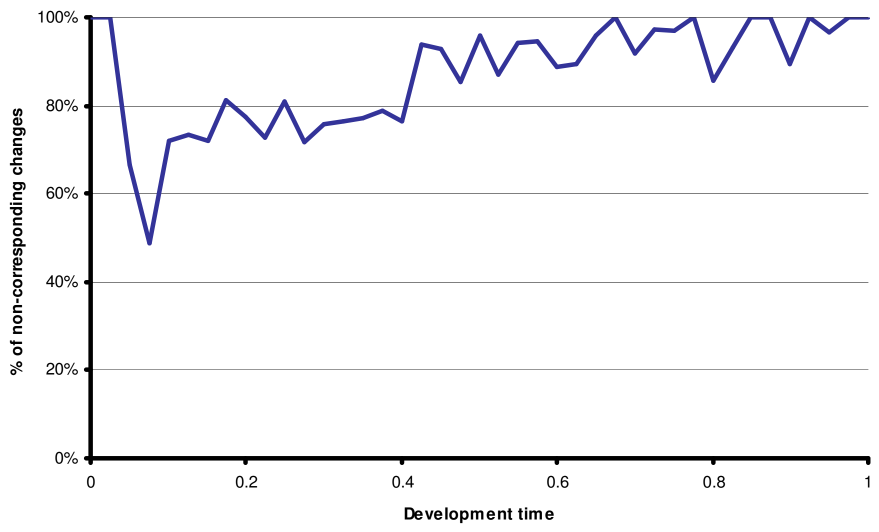


Figure 10 - Evolution of non-corresponding changes in Zope

Of the four projects, only Trac shows a very slight decline in the percentage of revisions that do not correspond to the structure of the product. Considering hypothesis 3 across all projects there is no evidence to support it. The amount of work that does not find structural correspondence does not appear to vary significantly with the different stages of the projects' development.

4.4.3 Hypothesis 4

Hypothesis 2 has shown that there is a large portion of tasks that make changes to groups of files without there being a structural connection between them and hypothesis 3 has shown that these tasks do not appear to be constrained to specific stages of the development of the product. Engineers are nevertheless making associations between files so that they can complete the tasks in hand. There is an implicit justification for selecting a group of files and this does not match what can be observed in the function call graph, an explicit representation of ties and dependencies. In hypothesis 2 the verification of the correspondence between what is changed and the inter-connectedness between those files is done at the moment the change is made. If the reasons that lead a programmer to associate different files take longer to manifest themselves in the function call graph, then the comparison initially done in hypothesis 2 may be off by time. This means that if we check later in the project's development for the connections between the files involved in a change, we might find that in their future state they do possess ties that establish the correspondence between the change and the product structure. If this is true, it can be useful for project managers as a way of predicting future dependencies and allocating resources to manage them. I therefore define hypothesis 4 as:

Hypothesis 4 - Work on a set of components without structural correspondence is prognostic of future structure.

The test of this hypothesis is done with the same elements from hypothesis 3. Each revision that changes multiple files is compared with the function call graph of the product at a future point in time. The last available revision was selected to be the point of comparison for each of the four projects.

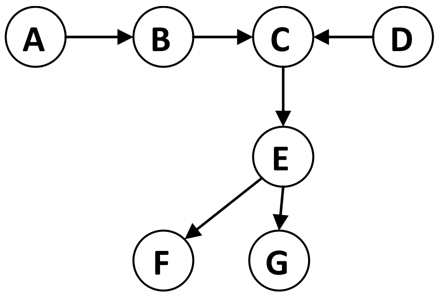
For each revision which didn't show structural correspondence initially, the results can belong to one of four scenarios:

- A revision may attain structural correspondence status later on only if the sufficient number of missing connections in the function call graph materialize in order to connect all the changed nodes.
- It can also be case that only some of the links materialize in a manner that is not enough to constitute a structural correspondence but nevertheless links do appear, also following the premise of hypothesis 4.
- Another case can be that the linkages previously found in the function call graph are no longer justified and thus removed.
- Finally, it is possible that no modification occurs.

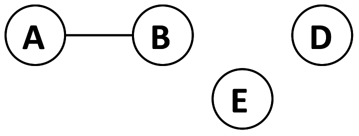
The first two cases support the hypothesis while the other two don't. Besides verifying if there is structural equivalence later in the project, the analysis will take into account the differences in the number of links and connectedness of the graphs between the two temporal representations. The analysis steps are as follows:

First, I calculate the minimum number of links needed for the original revision to have structural correspondence. This is equal to the number of connected subgraphs plus the number of isolated nodes minus 1 when the nodes from the revision are taken from the function call graph at the time of the revision.

The following example demonstrates this calculation. From the simple call graph below, a revision makes changes to files A, B, D and E.



The resulting graph considering the files changed contains one subgraph (A and B) and two isolated nodes (D, E) as illustrated in the following figure.



Minimum number of links necessary to obtain connected graph = Subgraphs + Isolated nodes -1 = 1+2-1 = 2.

This same calculation is then repeated for the files changed in the revision but taking into consideration the final function call graph. The difference in number of necessary links will indicate if the product structure in the final state corresponds more or less to the revision associations than at the revision time. A decrease in the number of necessary links indicates closer correspondence while an increase shows less. For all the revisions of the four different projects the results are:

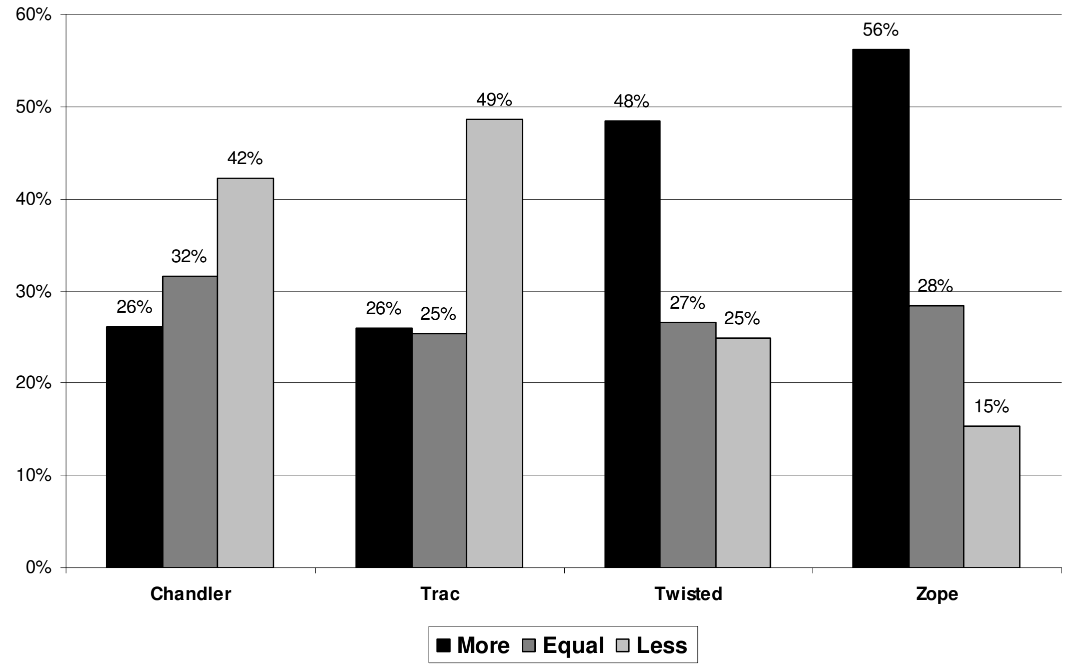


Figure 11 - Percentage of changes that become more or less connected over time

From the results it is possible to see that in projects Twisted and Zope, the most common observation shows the creation of structure, supporting the hypothesis, but the number of revisions that do not become more structurally correspondent is 52 and 43% respectively. In the remaining two projects the results are opposite, with the percentage of revisions that are less structurally correspondent with the final function call graph dominating the other two. This evidence does not support hypothesis 4 and the associations made between different product components do not seem to indicate the future presence of structure in between them.

4.4.4 Hypothesis 5

One of the ideas spawned from the results of testing hypothesis 2 was that depending on where work was being conducted, the effort of exploring a novel design to include additional features may not allow strong ties to be formed. The test of hypothesis 4 showed that the connectivity between files associated in a revision does not always increase over time. It remains to be shown where in the network work focuses on.

As seen from the literature review, in open source software development contributors prioritize and elect to work on the issues that are most promising and relevant to them. Their motivation may be to add a new feature or to fix an important defect. In terms of architecture, they can follow a modular or integral architecture. Previous studies show that open-source tends to organize into modular architectures (MacCormack, Rusnak & Baldwin 2006). What hasn't been observed is where, in the structure of a product and at a given point in time, do programmers make changes and how does the location of work evolve throughout the project's lifetime.

In a highly modular system, developers can theoretically focus first on the core components and then progress into the development of the peripheral modules. Or they can distribute load evenly across modules and develop them in parallel. Or they can first design the interfaces between modules before these are implemented. Or modules emerge later in the life of a project as complexity increases and an integral architecture becomes unmanageable and is purposefully broken down into modules.

These and other valid strategies (Baldwin & Clark 2000) allow programmers to focus on one part or another of the product to work on.

In an environment where there is little coordinated action on where work should focus, where do programmers elect to make their contribution? If there is a typical location or progress of work location in the lifetime of a project and it is possible to observe it against the current project it would be possible to identify performance deviations, guarantee that resources are available to work on the most important or critical components or attempt to redirect effort that neglects a portion of the product.

As also seen before, software products have highly connected and transitively dependent components which implement the desired functions and it is possible to extract this structure by constructing the function call graph. When studying this graph of interconnections between components, the betweenness centrality metric is the most adequate option as it incorporates the dependency nature of the paths between all the software files. Files that exhibit higher betweenness centrality are deemed as part of the core and more important to the system as their loss or reduced performance would propagate and impact the system more than a periphery and low betweenness centrality file.

From this, hypothesis 5 is generated as:

Hypothesis 5 - Central components are worked on most and are constantly being revised

To test this hypothesis I will use the same four projects and their function call graph through the projects' lifetimes. At each revision the call graph is built and analyzed using the betweenness centrality metric. This gives a list of all the nodes in the graph and their respective betweenness centrality measure. The nodes are ordered by their betweenness centrality value and those which represent the files changed in this revision are selected from this list, with their values and relative order position. The total number of existing nodes in this revision is also recorded into a file. This file is built with:

Revision number,

Node name,

Node rank in order,

Total number of nodes,

Betweenness centrality value for node

The process is repeated for all the other revisions in a project and then for all the projects. The results are then parsed in order to be visually analyzed. The value of the betweenness centrality is a normalized value between 0 and 1 (Wasserman & Faust 1994) which depends on the number of nodes in the measured network. The configuration of this network can change from revision to revision and each of the four projects sees variation in the number of nodes in the network as illustrated in Figure 12 with normalized values for project duration and number of files.

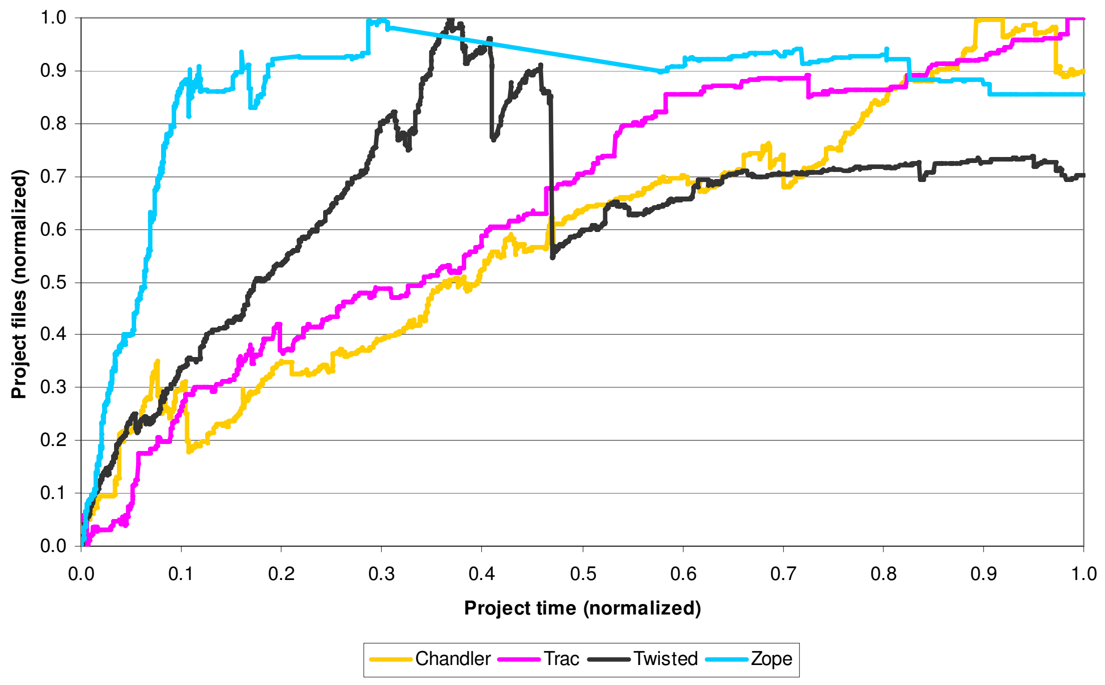
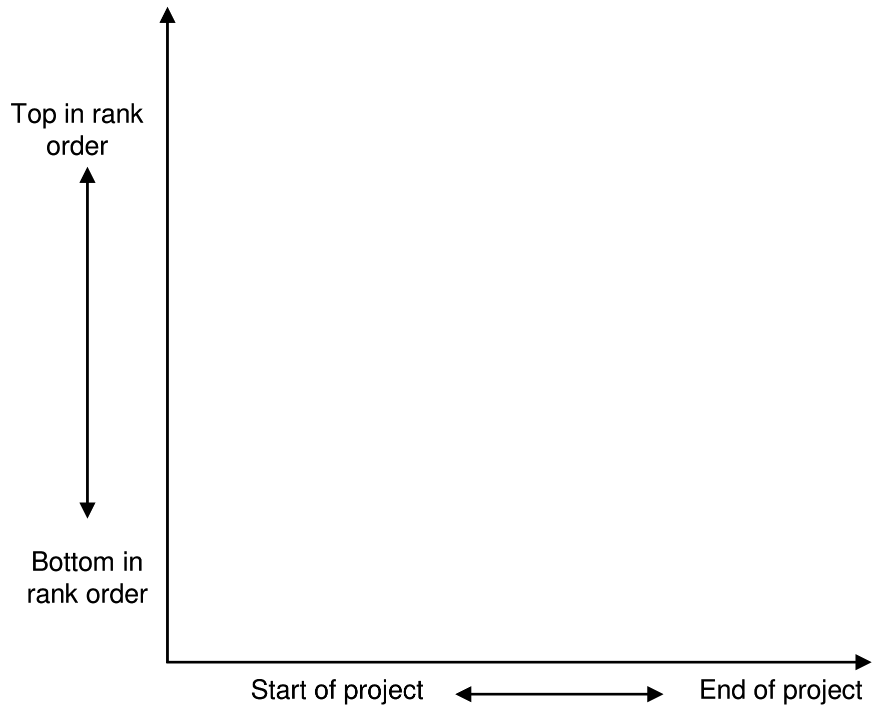


Figure 12 - File growth over project duration (normalized values)

A simple comparison of betweenness centrality measures between revisions would not be an apples-to-apples comparison since the underlying networks are themselves different. Betweenness centrality is also a metric that tries to measure the importance of a node and its connections to the other nodes in a network. Importance is a relative measure and with an available quantifier it is possible to order different values and determine which nodes are more or less important. For this reason I included the rank order of the nodes. The rank order normalizes the values across revisions. It determines the importance of the nodes relative to other nodes in their specific revision but with a value that can be used to compare with the other revisions.

It is now possible to compare these relative values throughout the duration of each project. The test of the hypothesis depends on the analysis of the rank order of the changed files in relation to the moment of the change. Time can be represented as clock time, using the date the change was submitted, or as sequential action, placing changes equally distant, acknowledging their sequence and ignoring the time differences in between them.

Since each revision corresponds to a data point coordinates and the different projects have several thousand revisions, I opted for a scatter plot with additive markers to represent all the data points. Each pair point is signaled with a marker at the respective coordinates. If another pair point has coordinates that fall near other points, the overlap in their markers will increase the darkness of that area. Several overlapping data points will create a significantly darker area than differently valued points. This allows creating a simulated heatmap of thousands of data points of activity compressed into a small scale and a small viewable area (Tufte 2001). By plotting the data points in a space defined by the two coordinates (time of change, betweenness rank order of change) we get:



For Chandler, the representation obtained is:

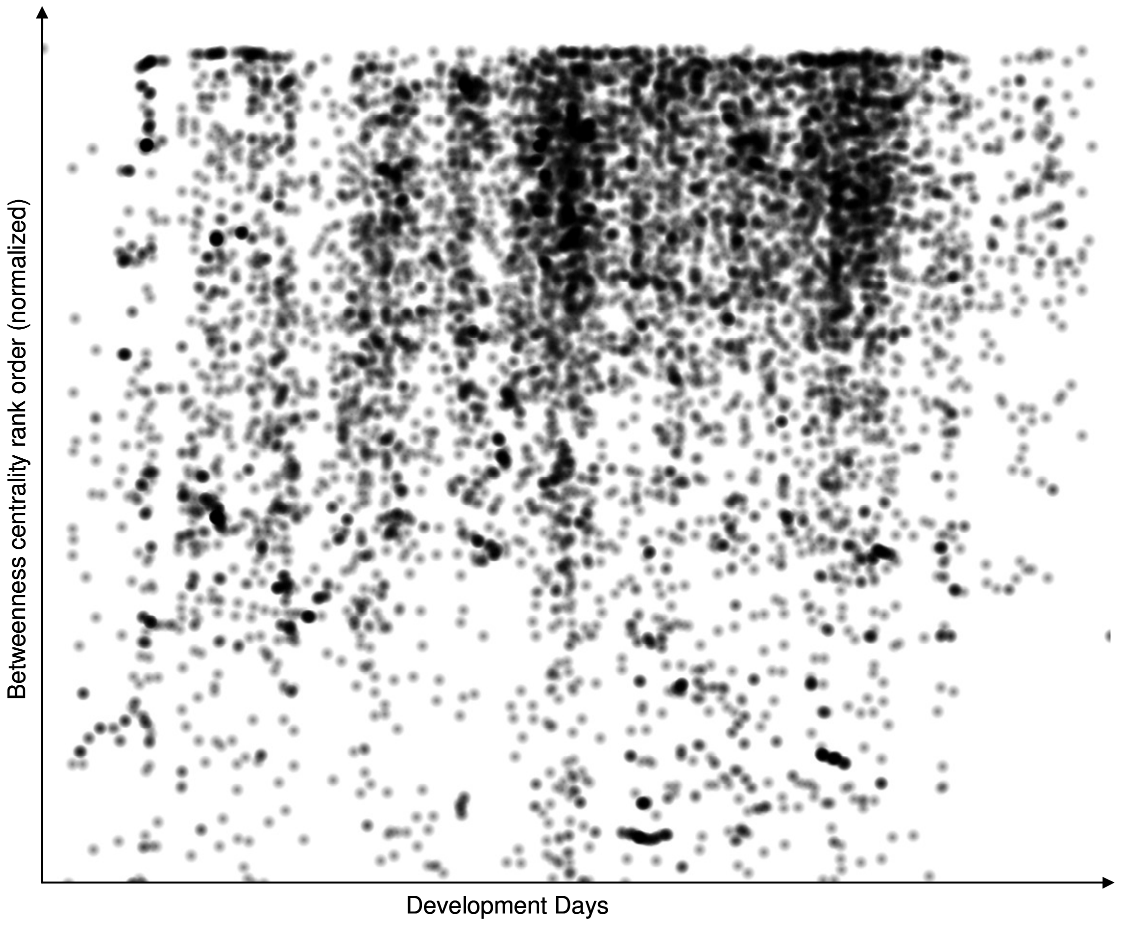


Figure 13 - Chandler betweenness centrality order of files changed over time

From this figure it is possible to observe that, at the start of the project, activity was spread over a wider range of portions of the product graph. As the project evolved, attention focused on a narrower group of files with higher betweenness centrality measures. Also noticeable are two darker areas which show bursts of activity. In the representation where the temporal spacing between revisions is removed, we can observe the following:

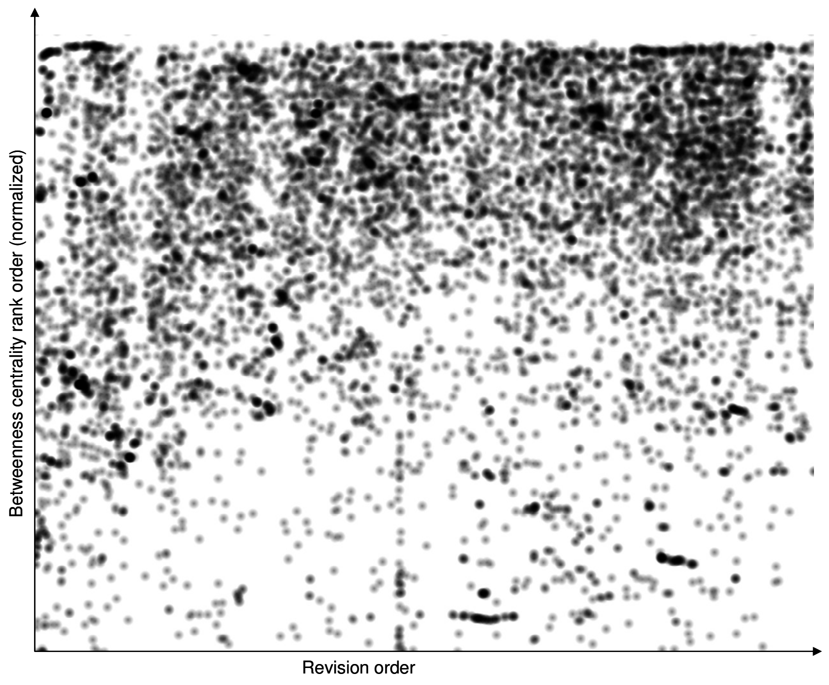


Figure 14 – Chandler betweenness centrality order of files changed by revision

Figure 14 illustrates the same work but separating each revision equally, not along a temporal dimension. This more clearly shows the progress from changing files which are generally disperse across the betweenness centrality rank to changing the files with higher betweenness centrality over the project's lifetime. Dividing the project in ten equal phases and calculating for each phase how much of the work falls in the top third of betweenness centrality rank we get:

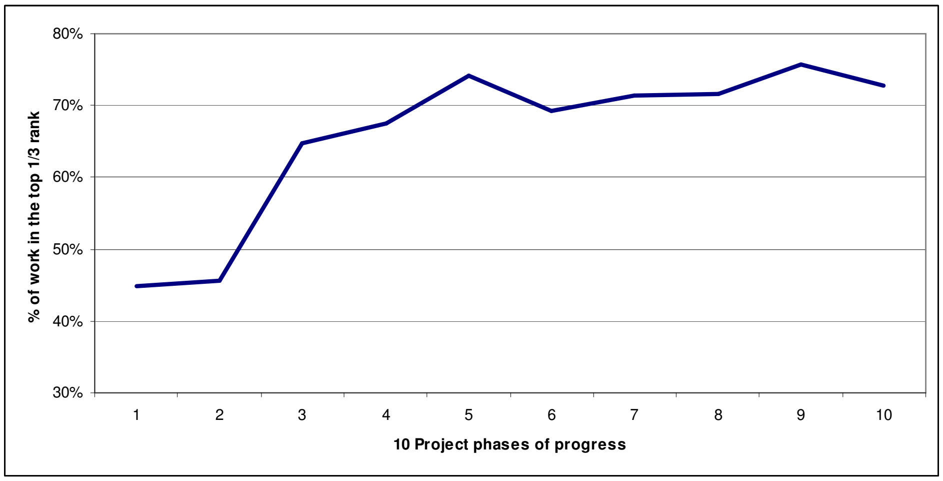


Figure 15 – Evolution of the percentage of work in the top third of betweenness centrality rank

The analysis is repeated for the other three projects and their map of work is represented in the following charts.

Project Trac

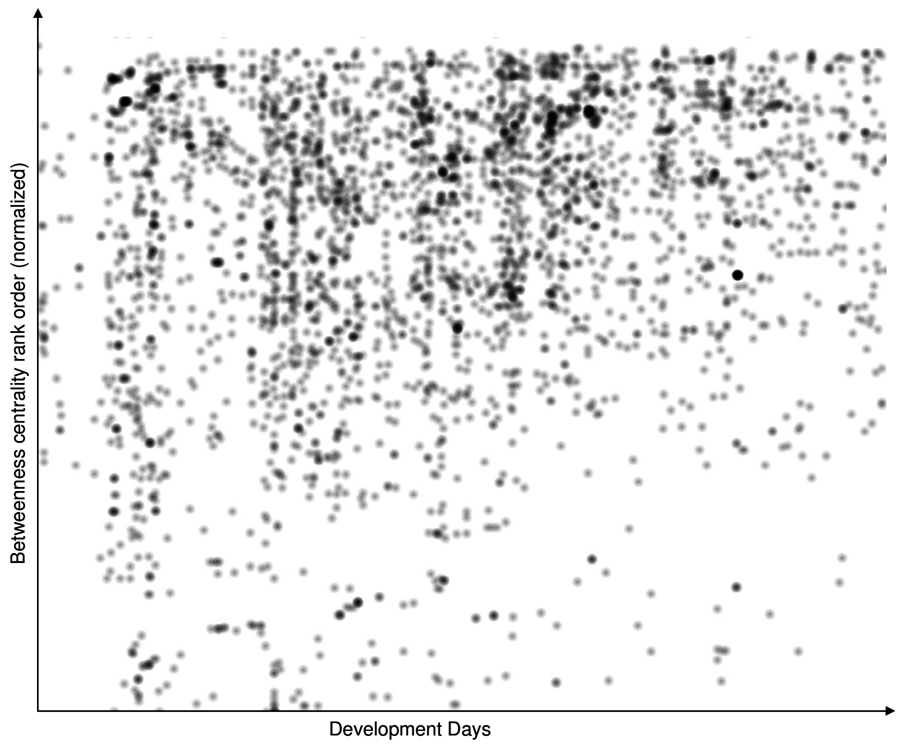


Figure 16 - Trac betweenness centrality order of files changed over time

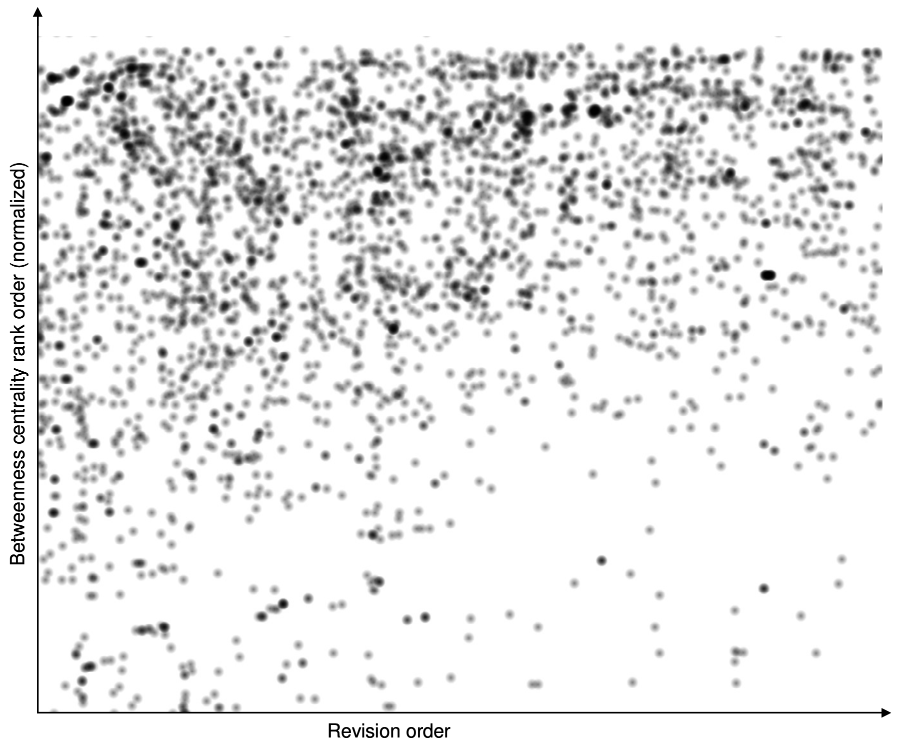


Figure 17 - Trac betweenness centrality order of files changed by revision

Project Twisted

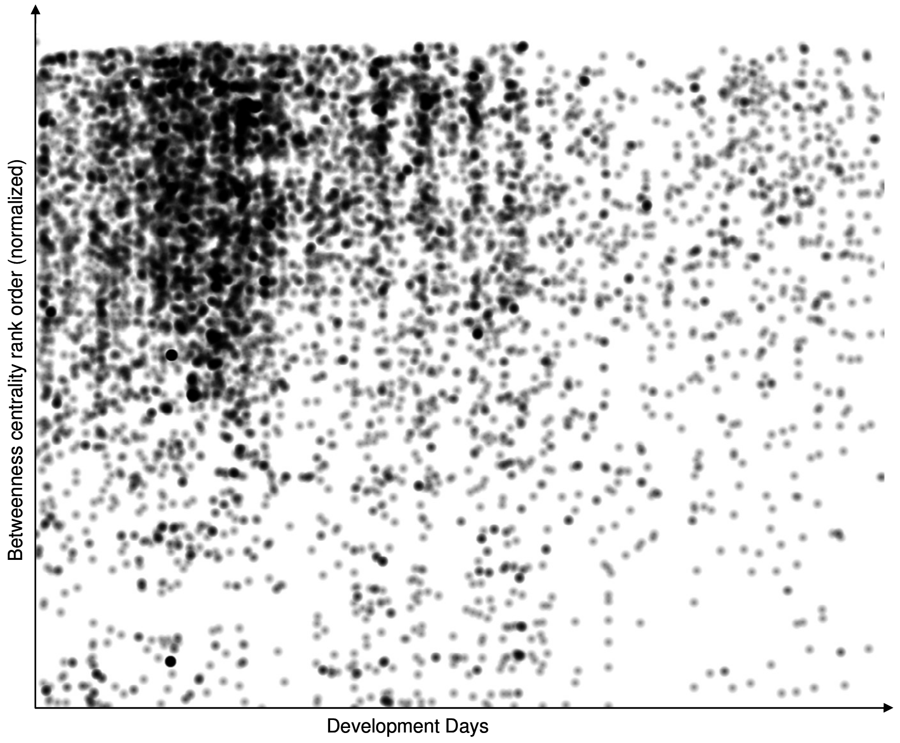


Figure 18 - Twisted betweenness centrality order of files changed over time

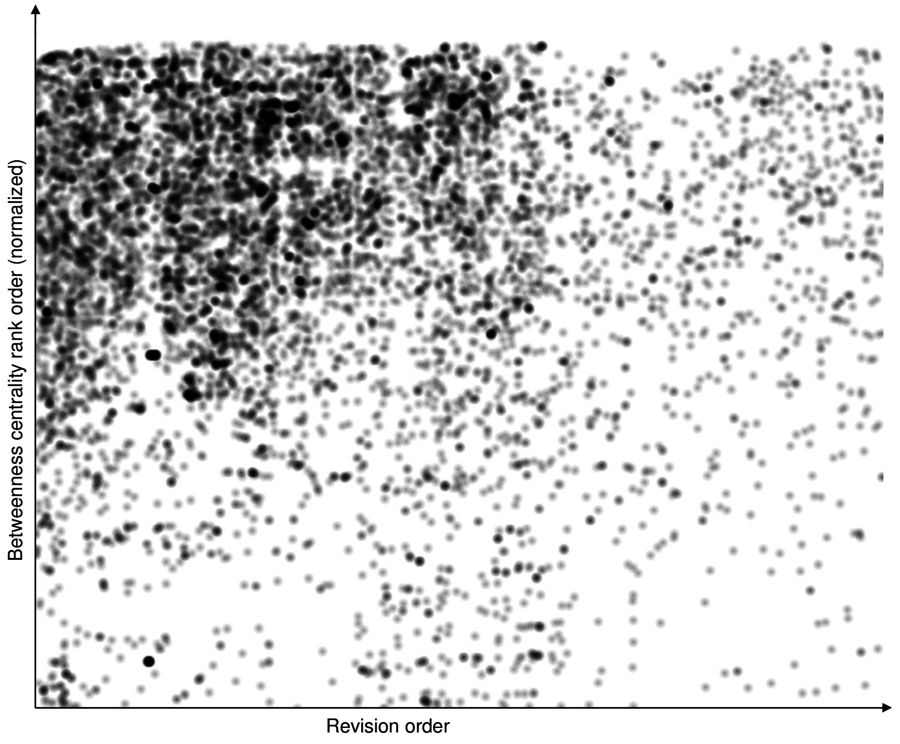


Figure 19 - Twisted betweenness centrality order of files changed by revision

Project Zope

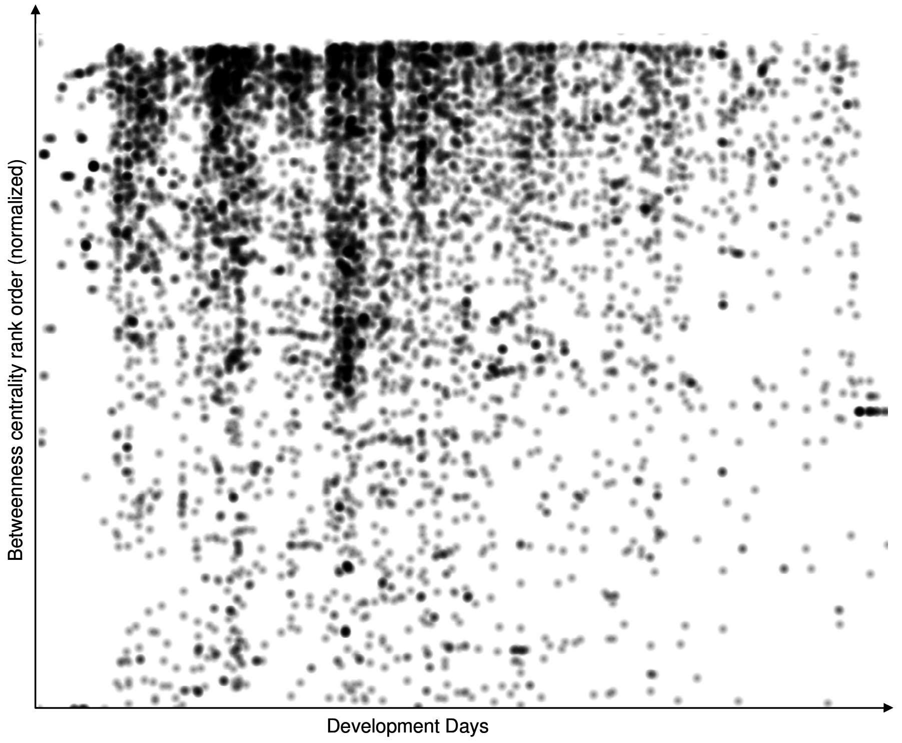


Figure 20 - Zope betweenness centrality order of files changed over time

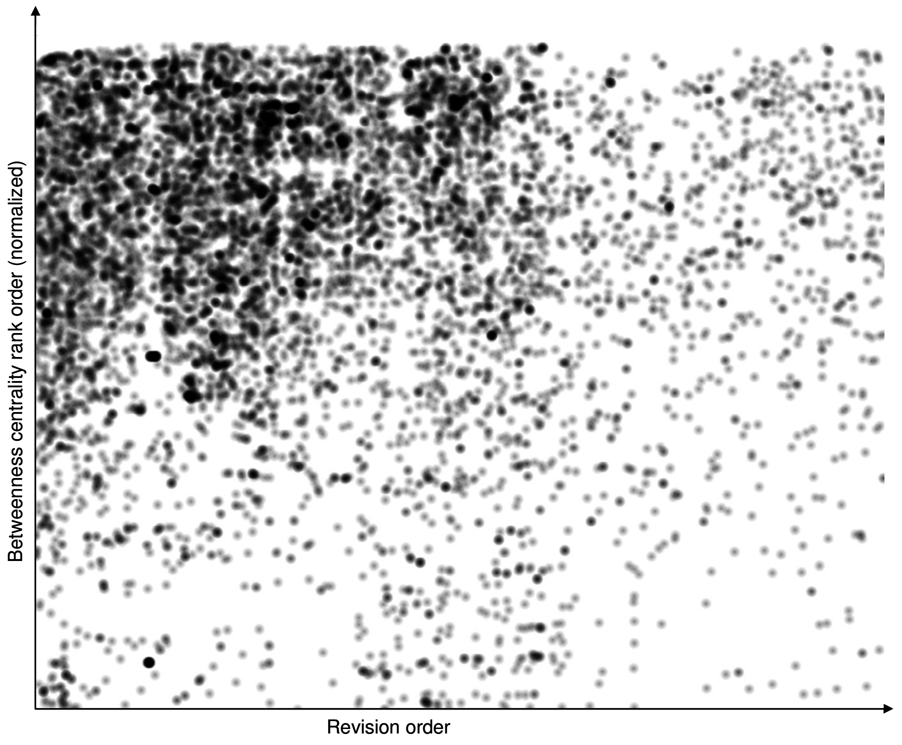


Figure 21 - Zope betweenness centrality order of files changed by revision

A fifth test was conducted in order to compare these results to what would happen when there is no rational or intentional decision involved in the selection of files to change. This test consisted of assigning a random set of files as those to be changed within a product with the same structure as the four projects. The baseline is that of a random selection made on the real product structures, not a random selection on a random graph.

The results are, using each product structure:

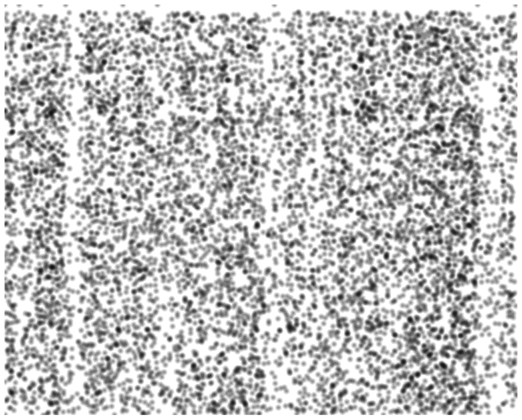


Figure 22 - Random based on Chandler structure

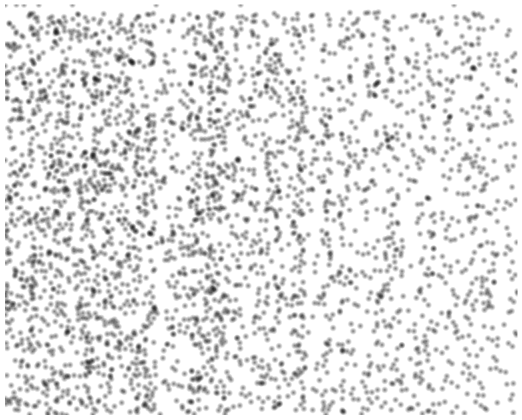


Figure 23 - Random based on Trac

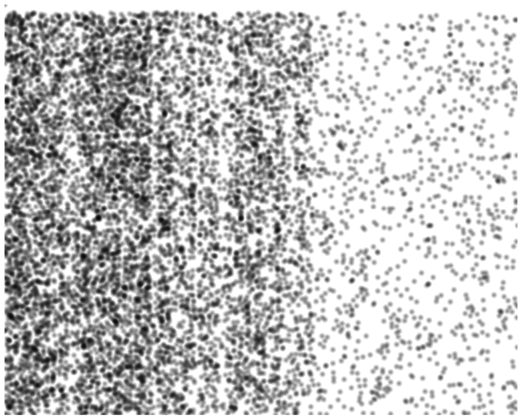


Figure 24 - Random based on Twisted

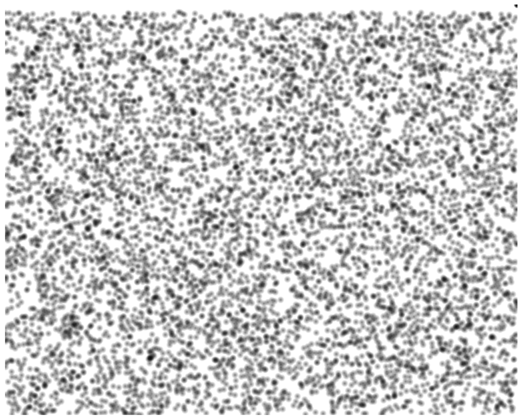


Figure 25 - Random based on Zope

By visually contrasting the above charts and those depicting the real projects (Figure 13, Figure 16, Figure 18 and Figure 20) it can be observed that those based on the project data are not similar to those generated by a random process, giving more relevance to their use as depictions of project progress and performance.

Mapping milestones

From the above charts it is possible to observe peaks of increased effort and periods with a reduced number of changes. I attempted to associate these with milestones in the different projects' execution. Projects of this nature typically have interim releases so the software can be tested. Every time a change is made the entire product can be tested and it is typical for all changes made in one day to be collected into a running version (compiled) so users can experiment using it. These versions are called the "nightlies" but, since there are created so often, testing them is usually not very deep and thorough. Once a sufficient set of features and fixes has been developed the project team may choose to "tag" the work at that point with an incremental version number. How much work is included and what number is chosen depends solely on the project team. A major rewrite of a program can see it progress from version 0.2 to 0.2.1 or a simple modification may be promoted as version 0.3 although there is some acceptance that "dot releases" (e.g., 0.1, 0.2, …) are those in which features have been added, sub-dot are fixes and integer releases are ready products (e.g 1.0, 2.0, …)

I collected information from each of the software repositories on the revision number and the date when the projects reached a dot release or an integer release. With the information available for Chandler, I projected the milestone occurrences onto the chart shown before, resulting in:

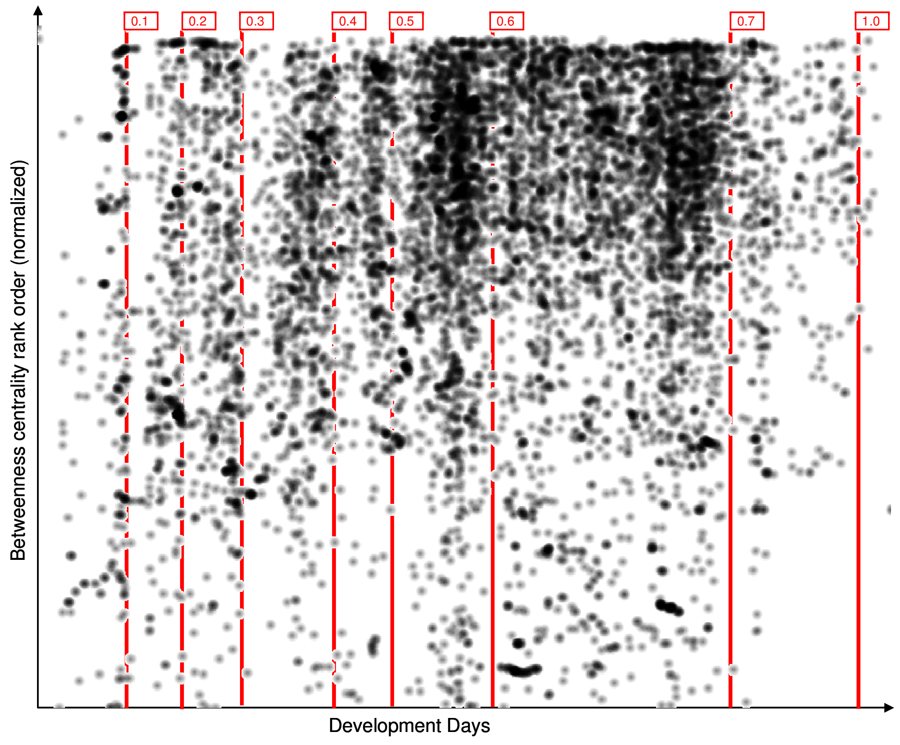


Figure 26 - Chandler development with milestones

From this chart it is possible to see that work ramps up preceding a milestone event and drops off after it. Chandler is a project that also has been object of an ethnographic study (Rosenberg 2008) which provides additional information describing it for part of its development from January 2003 through December 2005. In this piece we can find some evidence suggesting that the project objectives were not clear and the early critical decisions had not been made but, nevertheless, the team had gone ahead and started programming.

From Rosenberg:

> "Chandler 0.2 was unveiled on September 25, 2003, close to a year after OSAF had first announced Chandler to the world. It arrived under a cloud. The few users who downloaded it and tried it out were surprised to see that it actually did even less than Chandler 0.1 had. It was like the shell of a structure that had been gutted and only partially rebuilt."

By February 2004:

> "The progress also meant that the developers in the Apps Group, who had been content up to this point to let the design team work at its own pace, began to get itchy for a more thorough and final roadmap of how Chandler should look and behave. (…) the programmers' hunger for information grew palpable. They needed details. They needed blueprints. They needed specifications— a word so vital to the work of programming that over time it has shed all but its first syllable, becoming the terse, irreducible specs."

When the design work made headway it finally became apparent to the team that the early objectives for the project were too ambitious:

> "Kapor reviewed a set of explanations for why Chandler was so late: They had underestimated the cost of the project's big ambitions. It had proved harder than expected to build an engineering organization."
>
> "We bit off more than we could chew. The product is going to be more vanilla than we had originally hoped and more similar to its predecessors."

By November 2004:

> "With Chandler 0.5, OSAF's leadership had deliberately scaled back their ambitions and aimed low. They would forget for the moment the promise of organizing the entire universe of data and enabling outside developers to extend the program in unexpected directions. The soul of Agenda was a beautiful thing, they told their team; but for now, could we please just build a working calendar? Yet not far into the 0.5 schedule, which began at the start of November 2004, it became obvious that even that modest goal was beyond reach."

After Rosenberg stopped following the Chandler project it still released versions 0.6 and then 0.7 in September 2007. In January 2008 Mitch Kapor left OSAF and the Chandler project and announced he would stop funding the project (Parlante 2008). After his departure very little more work was done and a few fixes were added to 0.7 and the product was released as Chandler 1.0 in August 2008.

The Chandler project can then be summarized as a project that:

1 - Started coding too soon and before there was a defined design

2 - Once a design objective had been established progress could be made

3 - Market support and motivation for the product started to erode and a last effort was made

4 - Mitch Kapor decides to abandon the project

5 - Minor changes are made to launch whatever existed as a finalized product

Plotting the other projects

I also attempted to plot similar charts for the other projects using the "dot release" data. Marking milestones on the charts of Trac doesn't show a clear relation between the intensity of work and the occurrence of milestones.

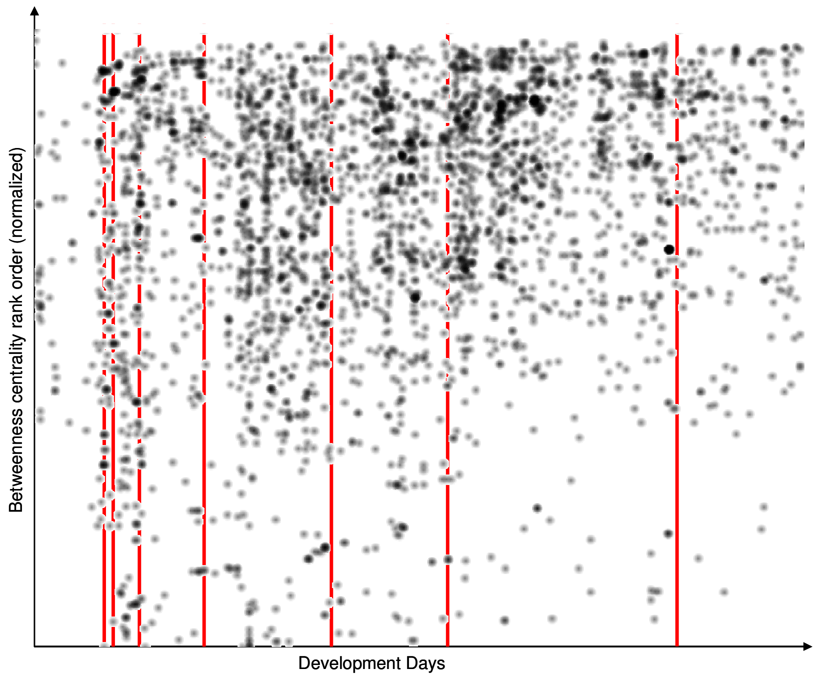


Figure 27 - Trac development with milestones

Project Zope's current repository only included milestones of a partial period of the project's development which preclude any further analysis beyond recognizing that there were notable periods of intense activity in the past and that such intensity hasn't repeated as can be seen by the lower density of dots on the right hand side of the chart.

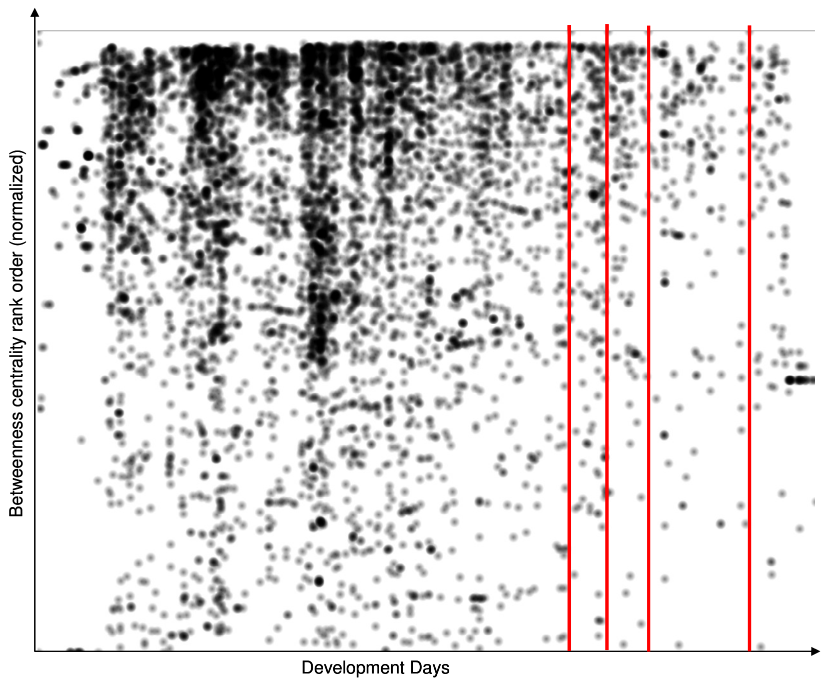


Figure 28 - Zope development with milestones (partial)

Project Twisted did not have any of this information available and from an interview conducted with one of Twisted's main project participants it was revealed that "because the lack of long term planning (…), you won't find any predictive milestones, since we never really used any." Nevertheless, from the Twisted chart of work over time (**Error! Reference source not found.**Figure 18) it is possible to see a cluster of activity early in the project's life and then a decline. This coincides with the description from the interview that "the assessment that the pace of commits has declined since 2003 sounds about right to me" as "Twisted became solid enough for [sponsor's] purposes, our development efforts shifted onto software outside of Twisted".

Histogram of changes (rank order placement)

Observing only the ranked importance of the nodes being changed and not taking into consideration the moment in which the change was made I can build a histogram of the betweenness centrality ranks. This is the same as building a histogram along the y-axis of the previous charts.

The objective of doing this is to verify how often the files with the highest betweenness centrality are being changed in comparison to the rest of the files. By simple observation of the previous charts I expect it to show that these are the files most frequently changed, and by calculating the histograms for each project I get:

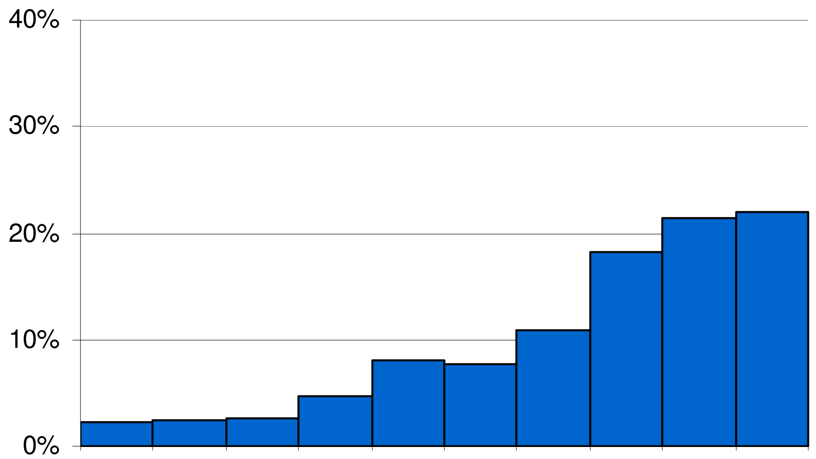


Figure 29 – Chandler project change betweenness centrality rank histogram (each bar represents a 10% interval)

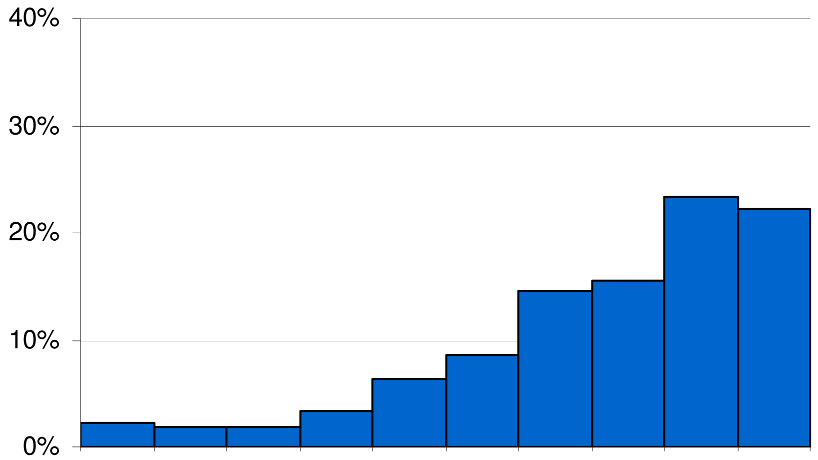


Figure 30 – Trac project change betweenness centrality rank histogram (each bar represents a 10% interval)

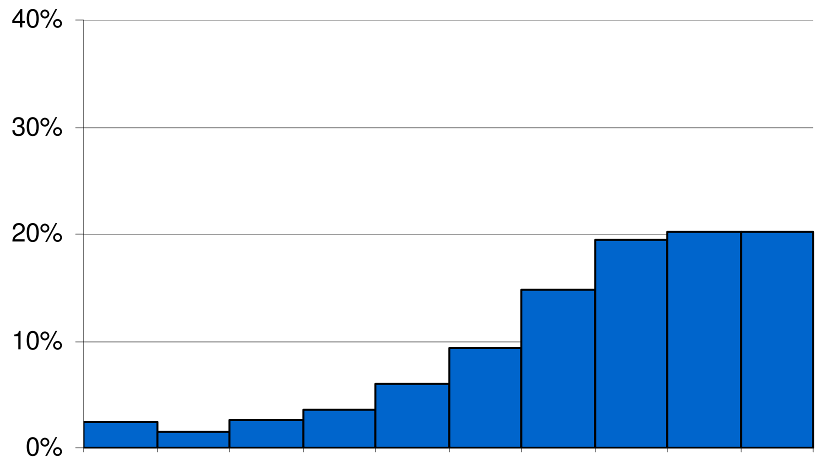


Figure 31 – Twisted project change betweenness centrality rank histogram (each bar represents a 10% interval)

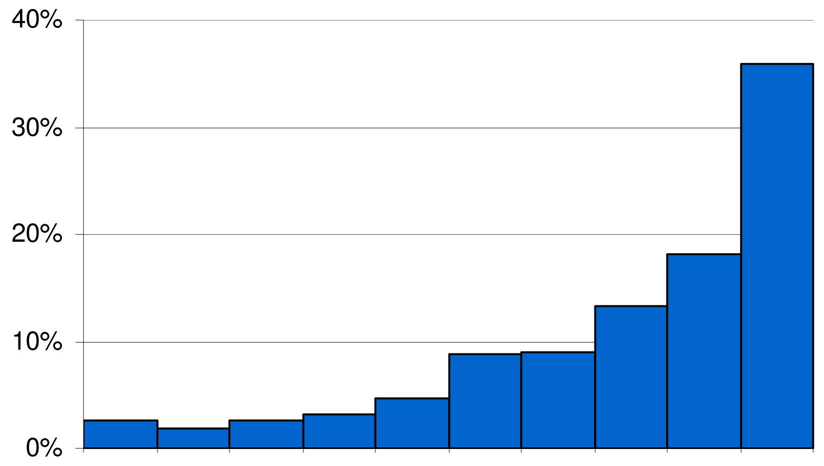


Figure 32 – Zope project change betweenness centrality rank histogram (each bar represents a 10% interval)

The analysis conducted on the data of the four projects in relation to hypothesis 5 has found evidence supporting that central components are worked on most and are constantly being revised. The use of a charted normalized rank of betweenness centrality over the execution of the project also proved useful to understand how the project is progressing. This type of representation can be translated into a tool for project managers to follow the progress of the projects they are managing.

4.4.5 Hypothesis 6

One issue related to hypothesis 5 is knowing what type of files occupy the higher betweenness centrality positions. As we have seen in Figure 12 the number of components that are part of these products is not constant so we are left to find out if the work that is done on the important files throughout the project are those which are created early or if new important components are constantly being created. I state Hypothesis 6 as:

Hypothesis 6 - Higher betweenness centrality components are those created early in a project's lifetime

In order to test this, I noted the revision in which a component was created and then noted every time a modification was made to it and the betweenness centrality rank value at that time.
By plotting a chart representing the age of components changed as the project evolves, it is possible to see how the distribution of work is made over new and older files. In this chart one axis represents time of development in a project and the other the relative age of component changed. The test of hypothesis 6 is only concerned with the files with higher importance, as measured through the betweenness centrality rank order so only the top third of the files in terms of importance are displayed.

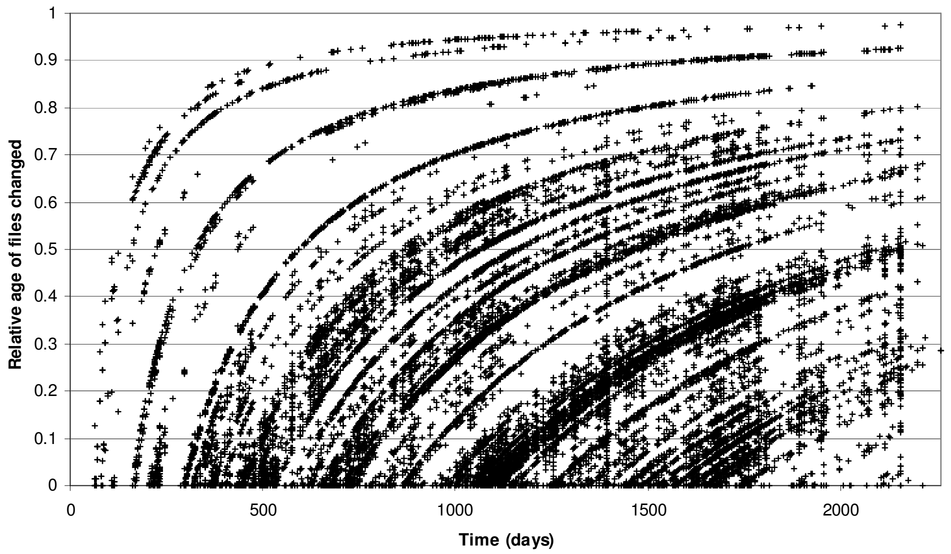


Figure 33 - Relative age of files changed over time in the Chandler project

Each mark on the chart illustrates a change made to a file and its coordinates reveal when the change was made (abscissa) and the age of the file (ordinate). The data points are not connected by lines but they are numerous enough to create the illusion of such in some places. These reveal that a file is being consistently being changed in a period of the development of the project. A sparser representation means the file is not altered as frequently as other files. Vertical bands appear when there is a revision or a set of revisions that change several files of different ages at the same time. From the analysis of the Chandler data in Figure 33 we see that some of the older files are continuously updated, as well as those created in the middle of the development effort. The files considered as most relevant in the Chandler project are continuously being changed, irrespective of age.

Plotting the same chart the other three projects we get:

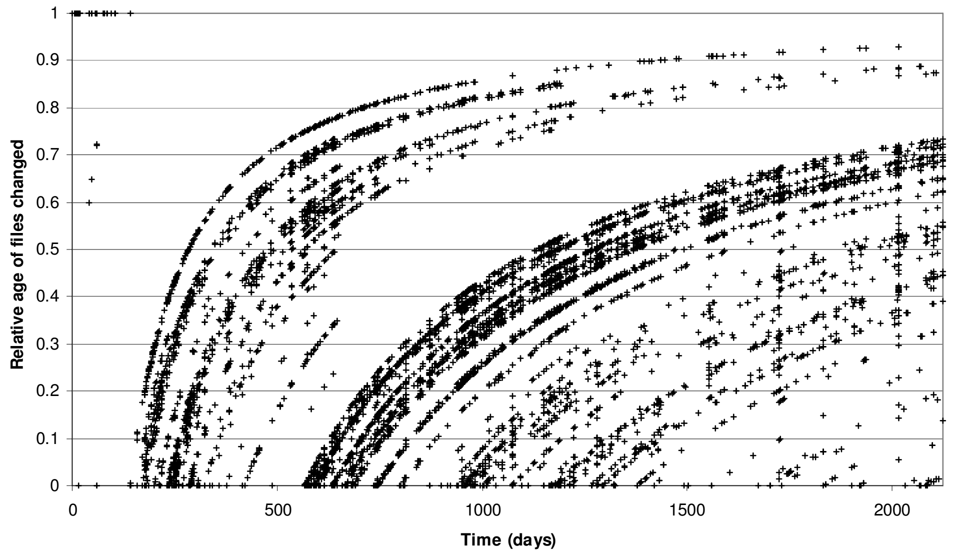


Figure 34 - Relative age of files changed over time in the Trac project

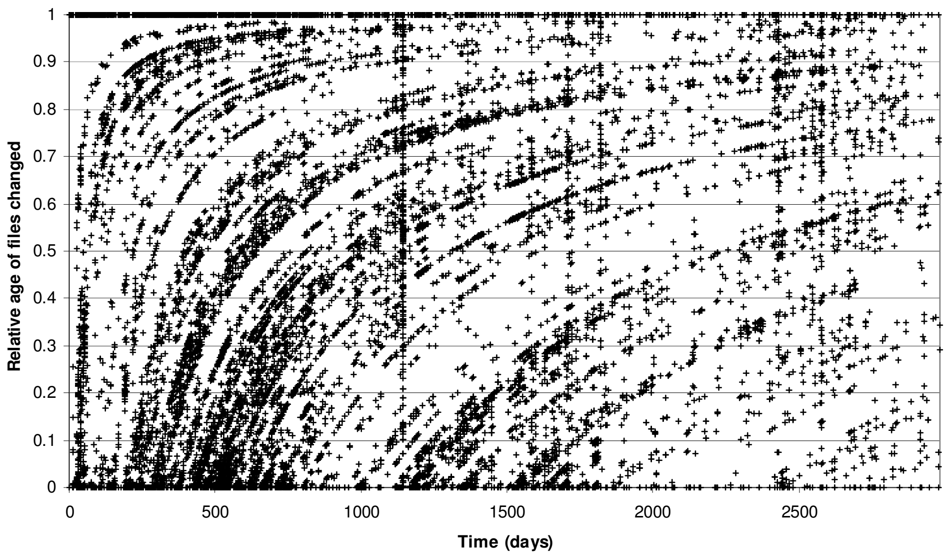


Figure 35 - Relative age of files changed over time in the Twisted project

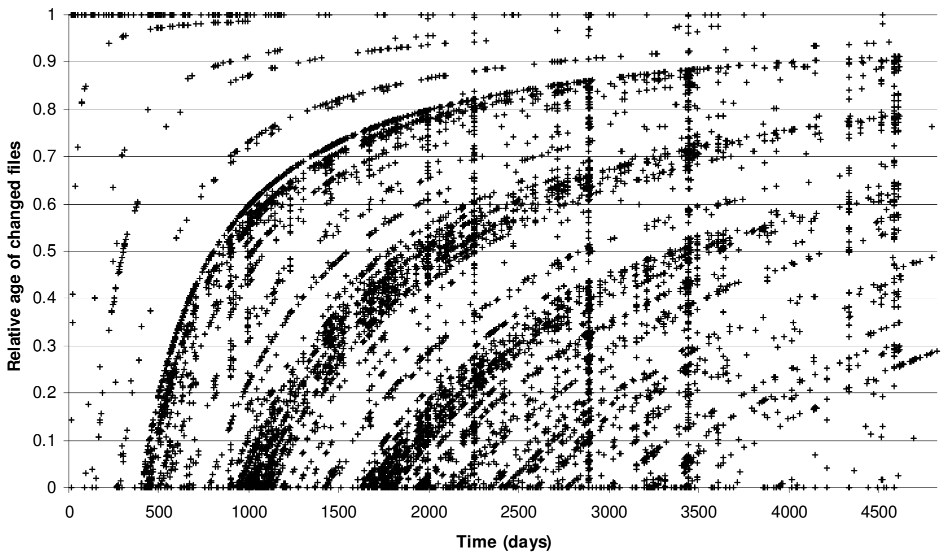


Figure 36 - Relative age of files changed over time in the Zope project

All projects reveal activity in files with high betweenness centrality measures that were created at different times throughout the development of the project. Hypothesis 6 is not confirmed.

4.4.6 External Validation

The results from the different hypothesis tested revealed some surprising figures, namely that associations between different files mostly do not follow the structure of the product, and do not even predict the existence of future structure between them.

The available academic literature on this topic is scarce and so a contrast with previous research is not possible. I interviewed four professionals in the field of software engineering and computer science research in order to get their commentary on the results that were obtained.
The four people were selected because of their experience in the software field and because they occupied different positions within the industry and/or academia and so could contribute with different points-of-view over the meaning of the data and tests.
The first expert is a professional program manager at one of the world's largest software companies and is responsible within a product line for a specific functionality of the product. As a program manager he has to evaluate functionalities to include in the product and how feasible they are to implement, plan the development and staff the team accordingly.
The second expert has more than a decade of experience as a software developer and program manager in one of the world's top mechanical engineering software tools.
The third expert is currently an academic researcher but previously had industry experience with the development of a mathematical software suite.
The fourth and final expert interviewed for this section of the thesis is a computer science researcher.

Each interviewee was asked to express their perspective on:

- why most of the changes happen in just one file
- what they guess the value calculated in hypothesis 2 would be
- why does the observed result have that value and if they accept it
- why do prognostic changes become less similar to the product structure

Regarding the number of changes affecting a single file, all experts thought that the high number of changes to single files is correct. Their commentary went on to suggest that this is due to the nature of the practice, where most changes would be "internal component optimization". Another reason is tied to the architecture where developers can make changes without changing a lot of files. In the opposite event "if most of the changes are done in more than one file simultaneously, then you probably messed up your architecture."

Finally, “the environment and practice of SW engineering tries to isolate developers so they don’t stomp each others toes” although the computer science researcher also added that he couldn’t relate this approach with a guarantee for a good design solution.

When asked to estimate the percentage of changes with structural correspondence one expert predicted values similar to those observed while another predicted a value in the neighborhood of 50%. The other two were not able to give an estimate. When confronted with the observed value all stated they weren’t very surprised by it and considered it plausible, with the fourth expert adding that it seemed like they weren’t maintaining good software practice.
Related to hypothesis 3, the second interviewee didn’t expect to see any temporal relationship while the third suggested “it depends on the time period in the project. In a mature project you would see little relationship between the function call graph and files being jointly committed.”, so with the later stages showing less structural correspondence than the earlier ones.

Finally, regarding the appearance of linkage in future structure or the use of changes as a prognostic of structure, experts agreed that there are scenarios in which connections can be created or not. They suggest that cases in which connections disappear (the “less” in the previous results) are most likely due to “code refactoring” or “in a phase where I’m growing and the functions get disaggregated” in which files are broken down into multiple other files and the functions making the connections moved onto these new files or that there are intermediate functions created in different files which replace the original connecting path, making it appear broken.

(this page intentionally left blank)

5. Conclusions

This thesis tested hypotheses related to how individuals operate in a product development environment, namely how they interact with other individuals and how they operate when facing the structure of the product.

The study on how individuals interact with one another focused on how each team member is of a different level of importance to each other and how, depending on the type of project being developed, these importance levels remain similar between projects. By comparing 13 design projects, it was found that for projects that have similar functional structure it is likely to find a similar connection between individuals. This means that when faced with a problem of similar nature, the profile of connections between individuals – which ones are more or less important – will also be similar.

Projects that are different in nature will exhibit very different associations of importance between participants.

These findings can have an impact in the practice of new product development as they allow project managers to project what the important associations between disciplines will be based on past projects, if they are similar in nature. With this information,

1- a project manager may be able to establish that participants engaged in connections that are deemed as essential or important be co-located or introduce more coordination events between them to improve synchronization.

2- Also, by using this information project managers can better monitor the execution of a project when observing the expected important channels. If no coordination actions happen along a

predicted important channel then the project manager can inquire and verify if there is a problem not being attended and address it.
3- A project manager can also, conversely, inspect channels that weren't predicted as important and that show activity.
4- Finally, those which have predicted and observed activity and those that do not have predicted nor observed activity are apparently behaving as expected.

The study on how individuals interact with the structure of a product showed that most individual work is localized and consists of internal improvement work. When work is done that requires simultaneous modifications of several components, it was found that individual behavior towards structural dependencies is not an expansion from the literature on teams. Teams typically coordinate along structural dependencies 80% of the time and individuals only operate this way 20% of the time. They create more associations other than structural between different components while working individually than when they require team interactions. This behavior seems to be consistent throughout the development of the projects and is not dependent on the design state of the product. It was also observed that individuals find and work on the most important components in a product and that work is conducted on components irrespective of their age in the system.
This thesis also described and presented results from an analysis process which can be built into a real-time progress tracking visualization tool for practitioners. By calculating the representations seen in Section 4.4.4, this tool would allow those involved in a product development project to monitor project activity levels and focus areas. It is possible to use this information in order to identify deviations and act to correct them. It is also straightforward to extract and monitor the contributions of individual profiles and verify if they are in line with what is expected.
Whenever faced with a deviation from what was established as typical performance, practitioners, before concluding there is an issue to be solved, should first consider the nature of what is being observe and if it makes sense in light of the current product objectives and actions requested to the development teams.

This thesis contributes directly to the stream of research of coordination in product development, namely on how:
- team communication channels are consistent over projects as long as they are similar in nature
- associations of product components made by individuals are not an extension of the associations made by teams
- individuals are able to find and work on the most important components of a project, as measured by the betweenness centrality metric

It also contributes to the practice with new methods to help those involved in large-scale complex product development filter the extensive work done by many individuals and find areas of possible intervention.

5.1.1 Future work:

Several paths of research can be taken in the future in order to extend the findings in this thesis namely:

The studies done on how individuals interact with the product structure were possible because software development practices already incorporate methods which record very detailed data. The nature of software allows it because software consists of text representations of algorithms, making it technologically easy to record and process. Although software development practices and literature has many common points with the classic product development literature, the generalizability of the findings in this thesis from the software projects is not straight forward. Software structures are limited by the logical representations in which the algorithms are built and a software component may connect to a much higher number of components than what is typically found in mechanical devices. This line of research would have much to gain if it could be replicated with products of different nature, such as mechanical or electronic products. The analysis of this type of products requires the use of different methods to extract the structure of the product as its design evolves as well as a wide access to each participant in the project's work data and it is not clear how such data collection could be done.

Other area which can be further developed relates to eliciting team connections from what each individual works on. From the work that is conducted by each individual it is possible to determine the areas of intervention on the system and how it evolves. By comparing these footprints between different individuals it is possible to measure how common the work each pair of participants is. If two individuals are working on a similar set of files then they should probably communicate with each other. A manager can verify if they are indeed coordinating, if they need better resources to do so or if they are aware of the actions of each other.

Using the same methods employed in this thesis, it would be possible to verify whether participants in a development effort are collectively ignoring important components and relate projects in which this happens to their progress and development performance.

These are some questions related to individuals and product development that can be explored. In order to do this, one would have to gain access to data not immediately available, such as communication data and project performance data. If those can also be captured then the ideas above can be further explored and can contribute to the academic literature as well as current practice.

Finally, we have seen that individuals in software development do not select a set of components to work on based on their dependencies. Still unsolved and not understood is determining what mechanism leads these individuals to make these associations between components.

The issues related to product development and the complexities associated with having many people working collaboratively in the design and development of new products is a very rich research field. Findings can not only improve how products and services are created but how teams and cooperative work can perform better. I hope the work that drove this thesis has been able to contribute and improve our understanding of these issues.

6. Bibliography

Alavi, M. et al. 2001. Review: Knowledge Management and Knowledge Management Systems: Conceptual Foundations and Research I... more. *MIS Quarterly*, **25**(1), pp. 107 - 136.

Allen, F.E. 1975. Interprocedural analysis and the information derived by it in *Programming Methodology*. Springer Berlin / Heidelberg, pp. 291-321.

Allen, T.J. 1984. *Managing the flow of technology : technology transfer and the dissemination of technological information within the R&D organization* 1st pbk. p. Cambridge, Mass.: MIT Press.

Ancona, D. et al. 2002. The Comparative Advantage of X-Teams. *MIT Sloan Management Review*, **43**(3), pp. 33-39.

Apache Subversion 2010. Subversion.

Avnet, M. 2009. Socio-cognitive Analysis of Engineering Systems Design: Shared Knowledge, Process and Product. *MIT - Engineering Systems Division*, p. 274.

Bagozzi, R.P. et al. 2006. Open source software user communities: A study of participation in Linux user groups. *Management Science*, **52**(7), p. 1099.

Baldwin, C.Y. et al. 2000. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press.

Baldwin, C.Y. et al. 1997. Managing in an age of modularity. *Harvard business review*, **75**(5), p. 84.

Boehm, B. et al. 2000. Spiral development: Experience, principles, and refinements.

Boehm, B. 1986. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, **11**(4), p. 14.

Borgatti, S.P. et al. 2009. Network analysis in the social sciences. *Science (New York, N.Y.)*, **323**(5916), pp. 892-5.

Braha, D. et al. 2004. Information flow structure in large-scale product development organizational networks. *Journal of Information Technology*, **19**(4), pp. 244-253.

Braha, D. et al. 2004. Topology of large-scale engineering problem-solving networks. *Physical Review E*, **69**(1), p. 16113.

Braha, D. 2002. Partitioning Tasks to Product Development Teams. *Proceedings of the ASME 14th International Conference on Design Theory and Methodology*.

Brannick, M.T. et al. 1997. *Team performance assessment and measurement : theory, methods, and applications*. Mahwah, N.J.: Lawrence Erlbaum Associates.

Bresnahan, T. et al. 2009. *Schumpeterian competition and diseconomies of scope; illustrations from leading historical*.

Brett, J. et al. 2006. Managing Multicultural Teams. *Harvard Business Review*, **84**(11), pp. 84-91.

Brown, S.L. et al. 1995. Product Development: Past Research, Present Findings, and Future Directions. *The Academy of Management Review*, **20**(2), pp. 343-378.

Cagan, J. et al. 2002. *Creating breakthrough products : innovation from product planning to program approval*. Upper Saddle River, NJ: Prentice Hall PTR.

Callahan, D. et al. 1990. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, **16**(4), p. 483–487.

Carlile, P.R. et al. 2002. A Pragmatic View of Knowledge and Boundaries: Boundary Objects in New Product Development. *Organization Science*, **13**(4), pp. 442-455.

Chiocchio, F. 2007. Project team performance: A study of electronic task and coordination communication. *Project Management Journal*, **38**(1), p. 97.

Christensen, C.M. et al. 2006. The Tools of Cooperation and Change. *Harvard Business Review*, **84**(10), pp. 73-80.

Clark, K.B. et al. 1991. *Product development performance : strategy, organization, and management in the world auto industry*. Boston, Mass.: Harvard Business School Press.

Colfer, L. et al. 2010. *The Mirroring Hypothesis : Theory , Evidence and Exceptions*.

Collberg, C. et al. 2003. A system for graph-based visualization of the evolution of software in *Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03*. New York, New York, USA: ACM Press, p. 77.

Collins, S.T. et al. 2008. Evaluating Product Development Systems Using Network Analysis. *Systems Engineering*.

Conway, M.E. 1968. How Do Committees Invent? *Datamation*, **14**(4), p. 28.

Cooper, R.G. 1993. *Winning at new products: accelerating the process from idea to launch* 2nd. Reading, Mass.: Addison-Wesley.

Creveling, C.M. et al. 2003. *Design for Six Sigma : in technology and product development*. Upper Saddle River, N.J.: Prentice Hall PTR.

Crowston, K. et al. 2002. Open source software projects as virtual organisations: competency rallying for software development. *IEE Proceedings - Software*, **149**(1), p. 3.

Crowston, K. et al. 2005. The social structure of free and open source software development. *First Monday*, **10**(2).

Cubranic, D. et al. 1999. Coordinating open-source software development in *IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999.(WET ICE'99) Proceedings*. IEEE, p. 61–66.

Cusumano, M.A. et al. 1998. *Thinking beyond lean : how multi-project management is transforming product development at Toyota and other companies*. New York: Free Press.

Deming, W.E. 2000. Introduction to a System in *The new economics: for industry, government, education*. Cambridge, Mass.: MIT Press.

Deming, W.E. 2000. *The new economics : for industry, government, education* 2nd. Cambridge, Mass.: MIT Press.

Drucker, P. 1988. The coming of the new organization. *Harvard business review*, **66**(1), p. 45–53.

Eppinger, S.D. et al. 1994. A model-based method for organizing tasks in product development. *Research in Engineering Design*, **6**(1), pp. 1-13.

Eppinger, S.D. et al. 2001. Patterns of product development interactions in *International Conference on Engineering Design*.

Eppinger, S.D. 1997. A planning method for integration of large-scale engineering systems in *International Conference on Engineering Design*., p. 199–204.

Free Software Foundation 2007. GNU General Public License.

Galbraith, J.R. 1973. *Designing Complex Organizations*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Gershenfeld, N.A. 2005. *Fab : the coming revolution on your desktop--from personal computers to personal fabrication*. New York: Basic Books.

Gerwin, D. et al. 2002. An Evaluation of Research on Integrated Product Development. *Management Science*, **48**(7), pp. 938 - 953.

Giffin, M. et al. 2009. Change Propagation Analysis in Complex Technical Systems. *Journal of Mechanical Design*, **131**(8), p. 081001.

Godfrey, M.W. et al. 2000. Evolution in open source software: a case study in *Proceedings of the International Conference on Software Maintenance.*, pp. 131-142.

Griffin, A. et al. 1992. Patterns of communication among marketing, engineering and manufacturing-A comparison between two new product teams. *Management Science*, **38**(3), p. 360–373.

Grove, D. et al. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, **23**(6), pp. 685-746.

Halloran, T. et al. 2002. High quality and open source software practices. *Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering*, p. 1–3.

Hauptman, O. et al. 1999. Managing integration and coordination in cross-functional teams: an international study of Concurrent Engineering product development. *R and D Management*, **29**(2), pp. 179-192.

Hemetsberger, A. et al. 2004. Sharing and Creating Knowledge in Open-Source Communities: The case of KDE

Hertel, G. et al. 2003. Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. *Research Policy*, **32**(7), pp. 1159-1177.

Howison, J. et al. 2004. The perils and pitfalls of mining SourceForge. *Proceedings of the International Workshop on Mining Software Repositories (MSR 2004)*, (April 2002), p. 7–11.

Kapor, M. 2010. Mitchell Kapor: Biography.

Katz, R. 1982. The Effects of Group Longevity on Project Communication and Performance. *Administrative Science Quarterly*, **27**(1), pp. 81-104.

Kelley, T. et al. 2005. *The ten faces of innovation : IDEO's strategies for beating the devil's advocate & driving creativity throughout your organization*. New York: Currency/Doubleday.

Kennedy, M.N. 2003. *Product development for the lean enterprise : why Toyota's system is four times more productive and how you can implement it*. Richmond, Va.: Oaklea Press.

Kim, B. et al. 2008. Cross-functional cooperation with design teams in new product development. *International Journal of Design*, **2**(3), p. 43–54.

Klein, G. et al. 1999. Distributed Planning Teams. *International Journal of Cognitive Ergonomics*, **3**(3), p. 0.

Klein, K.J. et al. 2004. How do they get there? An examination of the antecedents of centrality in team networks. *Academy of Management Journal*, **47**(6), p. 952–963.

Kleinbaum, A.M. et al. 2008. *Communication (and Coordination?) in a Modern, Complex Organization*.

Knowledge @ Wharton 2006. Is Your Team Too Big? Too Small? What's the Right Number? *Knowledge @ Wharton*, pp. 1-4.

Koch, S. et al. 2002. Effort, co-operation and co-ordination in an open source software project: GNOME. *Information Systems Journal*, **12**(1), pp. 27-42.

Koch, S. 2005. Evolution of Open Source Software Systems–A Large-Scale Investigation in Sotto, M. et al., *Proceedings of the First International Conference on Open Source Systems*. Genova, pp. 148-153.

Kraut, R.E. et al. 1995. Coordination in software development. *Communications of the ACM*, **38**(3), pp. 69-81.

Krishnamurthy, S. 2002. Cave or community?: An empirical examination of 100 mature open source projects. *First Monday*, **7**(6).

LaMantia, M.J. et al. 2007. *Evolution Analysis of Large-Scale Software Systems Using Design Structure Matrices and Design Rule Theory.*

Lakhani, K. et al. 2003. Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects.

Lanza, M. 2001. The evolution matrix: recovering software evolution using software visualization techniques in ACM New York, NY, USA, pp. 37-42.

Laplante, P.A. 2000. Call Graph in Laplante, P. A. , *Dictionary of Computer Science, Engineering and Technology.*, p. 63.

Lawrence, P.R. et al. 1967. Differentiation and Integration in Complex Organizations. *Administrative Science Quarterly*, **12**(1), pp. 1-47.

MacCormack, A. et al. 2008. *Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis.*

MacCormack, A. et al. 2006. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Management Science*, **52**(7), pp. 1015-1030.

MacCormack, A. et al. 2007. *The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry.* Harvard Business School.

Madey, G. et al. 2002. The open source software development phenomenon: An analysis based on social network theory in *Americas Conference on Information Systems (AMCIS2002).*, pp. 1806-1813.

Malone, T. 1987. Modeling Coordination in Organizations and Markets. *Management Science*, **33**(10), pp. 1317-1332.

McGrath, M.E. 2004. *Next Generation Product Development : How to Increase Productivity, Cut Costs, and Reduce Cycle Times*. McGraw-Hill.

McGrath, M.E. 1996. *Setting the PACE in Product Development, A Guide to Product and Cycle-time Excellence (Paperback)*. Butterworth-Heinemann.

Mockus, A. et al. 2000. A case study of open source software development: the Apache server in ACM Press New York, NY, USA, pp. 263-272.

Mockus, A. et al. 2002. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, **11**(3), pp. 309-346.

Mockus, A. et al. 2003. Understanding and predicting effort in software projects in, pp. 274-284.

Mockus, A. et al. 2002. Why Not Improve Coordination in Distributed Software Development by Stealing Good Ideas from Open Source? in *Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering.*, pp. 2001-2003.

Morelli, M.D. et al. 1995. Predicting technical communication in product development organizations. *IEEE Transactions on Engineering Management*, **42**(3), pp. 215-222.

Morgan, J.M. et al. 2006. *The Toyota Product Development System: Integrating People, Process And Technology*. Productivity Press.

Murphy, G. et al. 1998. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **7**(2), p. 158–191.

Nakakoji, K. et al. 2002. Evolution patterns of open-source software systems and communities in *Proceedings of the international workshop on Principles of software evolution - IWPSE '02*. ACM New York, NY, USA, pp. 76-85.

Olson, E.M. et al. 2001. Patterns of cooperation during new product development.pdf. *Journal of Product Innovation Management*, (18), pp. 258-271.

Pahl, G. et al. 1996. *Engineering design : a systematic approach*. London ; New York: Springer.

Parlante, K.C. 2008. OSAF Transitions.

Paulk, M. 2001. Extreme programming from a CMM perspective. *IEEE software*, **18**(6), pp. 19-26.

Rich, B. et al. 1996. *Skunk Works: A Personal Memoir of My Years of Lockheed*. Back Bay Books.

Roberts, J. et al. 2006. *Understanding the Motivations, Participation and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects*.

Rosenberg, S. 2008. *Dreaming in code*. Three Rivers Press.

Ruef, M. et al. 2003. The Structure of Founding Teams: Homophily, Strong Ties, and Isolation among U.S. Entrepreneurs. *American Sociological Review*, **68**(2), pp. 195 - 222.

Ryder, B. 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, **75**(3), p. 216–226.

Schein, E.H. 1996. Three Cultures of Management: The Key to Organizational Learning. *MIT Sloan Management Review*, **38**(1), pp. 9-20.

Schneider, S.C. et al. 2002. *Managing Across Cultures* 2nd. Prentice Hall.

Schrage, M. 1995. *No more teams! : mastering the dynamics of creative collaboration* 1st Curren. New York: Currency Doubleday.

Sommerville, I. 2006. *Software Engineering: (Update) (8th Edition)*. Addison Wesley.

Sosa, M.E. et al. 2004. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. *Management Science*, **50**(12), pp. 1674-1689.

Sosa, M.E. 2008. A structured approach to predicting and managing technical interactions in software development. *Research in Engineering Design*, **19**(1), pp. 47-70.

The Bugzilla Team 2010. Life Cycle of a Bug. *The Bugzilla Guide*.

Tigris 2010. TortoiseSVN.

Tomayko, J. et al. 2004. *Human Aspects of Software Engineering (Electrical and Computer Engineering Series)*. Charles River Media.

Torvalds, L. 2010. C++ productivity email thread.

Trac Project 2010. Trac Users.

Tufte, E.R. 2001. *The Visual Display of Quantitative Information, 2nd edition*. Graphics Press.

Ullman, D. 2002. *The Mechanical Design Process*. McGraw-Hill Science/Engineering/Math.

Ulrich, K.T. et al. 2000. *Product design and development* 2nd. Boston: Irwin/McGraw-Hill.

Wagner, D. et al. 2001. Intrusion detection via static analysis in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE Computer Society, pp. 156-168.

Wasserman, S. et al. 1994. *Social Network Analysis: Methods and Applications*. Cambridge University Press.

Womack, J.P. et al. 1990. *The machine that changed the world*. Rawson Associates New York.

Zope.org 2010. What is Zope?

von Hippel, E. 1990. Task partitioning: An innovation process variable. *Research Policy*, **19**(5), pp. 407-418.

von Krogh, G. et al. 2003. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, **32**(7), pp. 1217-1241.

von Krogh, G. et al. 2006. The promise of research on open source software. *Management science*, **52**(7), p. 975.

7. Appendix

03-changeLog-Parser.py

```
'''
Goes through a file with the SVN log and extracts data from records
and uploads it to tables in a MySQL database.

Data collected:
- Revision number
- Author
- Date
- Message
- Files changed
- Type of change

NOTE: add an empty line at the end of the changelog file to guarantee that
the readline loop runs one more time and empties everything it has into the database

'''

import re
import MySQLdb

# Open connection to database
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db="projectname")
dbcursor = db.cursor()

#open file
#path = '../Projects/Chandler/SourceCode/changelog.txt'
#path = '../Projects/Zope/changelog.txt'   #Done. Sept 8 2009
#path = '../Projects/Plone/changelog.txt'  #Done. Sept 8 2009
#path = '../2. Projects/Twisted/changelog.txt' #Attention with ERROR with 'Author: ' in the
middle of the message

fileHandle = open ( path, 'r')

fileList = fileHandle.readlines()

#initialize variables
RevisionCount = 0
inRevision = 0
inChanges = 0
inMessage = 0
ChangesVector = []
Revision = '' # 0= outside, 1= in a revision section, 2= done with revision
```

```
    Date = ''
    Author = ''
    MSG = ''

    for fileLine in fileList:

        #A test performed on previous line would give access to this area. See below for the test.
        if inChanges ==1 and (re.match('(Revision: .*)',fileLine)is None):
            change = fileLine.split(' : ')
            if len(change)>1: #this tests if the line has the two elements. this makes it skip the empty line between two revisions.
                #print 'INSERT INTO rev_files (commitNum, changeType, file) VALUES ("'+str(RevisionNumber)+'","'+change[0]+'","'+change[1].strip()+'")'
                dbcursor.execute('INSERT INTO rev_files (commitNum, changeType, file) VALUES ("'+str(RevisionNumber)+'","'+change[0]+'","'+change[1].strip()+'")')

        #A test performed on previous line would give access to this area. See below for the test.
        if inMessage ==1:
            if (re.match('(----)',fileLine)is not None):
                inMessage = 0
                #message is over
                if inRevision == 2:
                    #print '\n\nRev '+ str(RevisionNumber) + ', Author: '+Author+', Date: '+Date+'\n'+MSG
                    MSG = MSG.replace('"','')
                    if len(MSG)>6000:
                        MSG = 'JOAO: Message too big. Check original changelog in '+path+' for complete message.'
                    #print 'INSERT INTO rev_commit (commitNum, author, date, message) VALUES ("'+str(RevisionNumber)+'","'+Author+'","'+Date+'","'+MSG+'")'
                    dbcursor.execute('INSERT INTO rev_commit (commitNum, author, date, message) VALUES ("'+str(RevisionNumber)+'","'+Author+'","'+Date+'","'+MSG+'")')
                    MSG=''
                    Author = ''
                    Date = ''
                    inRevision = 0
            else:
                MSG = MSG + fileLine.strip()

        #check if we started a new Revision
        fromData = re.match('(Revision: .*)',fileLine)
        if (fromData is not None):
            # we are done with previous changes
            inChanges=0

            #start processing this revision
            inRevision =1
            RevisionCount += 1

            #Get the Revision number:
            RevisionNumber = fromData.group(0).lstrip('Revision: ')

        #get the Author
        fromData = re.match('(Author: .*)',fileLine)
        if (fromData is not None):
            Author = fromData.group(0).lstrip('Author: ')

        #get the Date
        fromData = re.match('(Date: .*)',fileLine)
        if (fromData is not None):
            Date = fromData.group(0).lstrip('Date: ')
            inRevision =2 #marks the end of the collection of info for the commit table

        #get the Message
        fromData = re.match('(Message:*)',fileLine)
        if (fromData is not None):
            inMessage =1

        #did we start the changes section?
        fromData = re.match('(----)',fileLine)
        if (fromData is not None):
            inChanges =1

    fileHandle.close()


    print 'Total number of revisions is ' + str(RevisionCount)
```

18-Checkout SVN.py

```
'''
Creates a local copy (checkout) of code from a SVN repository which can be
local (created before using svnsync) or remote.

This first chunk here is copied from
http://pysvn.tigris.org/svn/pysvn/trunk/pysvn/WorkBench/Source/wb_main.py

It is required to avoid a locale error when running on Linux. It was not
needed just to run in Windows.
Windows, on the other hand, can't deal with two files with same name but
that differ in case. For example: 'file' and 'FILE' are two different
things in UNIX but not in Windows.

So, now we are running the code in Linux to avoid the naming problem, but
had to add the locale code...
'''

import sys
import locale
import os

def initLocale():
    # init the locale
    if sys.platform == 'win32':
        locale.setlocale( locale.LC_ALL, '' )

    else:
        if 'LC_ALL' in os.environ:
            try:
                locale.setlocale( locale.LC_ALL, os.environ['LC_ALL'] )
                return
            except locale.Error:
                pass

        language_code, encoding = locale.getdefaultlocale()
        if language_code is None:
            language_code = 'en_US'

        if encoding is None:
            encoding = 'UTF-8'
        if encoding.lower() == 'utf':
            encoding = 'UTF-8'

        try:
            # setlocale fails when params it does not understand are passed
            locale.setlocale( locale.LC_ALL, '%s.%s' % (language_code, encoding) )
        except locale.Error:
            try:
                # force a locale that will work
                locale.setlocale( locale.LC_ALL, 'en_US.UTF-8' )
            except locale.Error:
                locale.setlocale( locale.LC_ALL, 'C' )

initLocale()
print 'Info: locale set to %r' % (locale.getlocale(),)

'''
This is my old code, before adding the Linux compatibility chunk.
'''

# References from
#    http://pysvn.tigris.org/docs/pysvn_prog_guide.html
#    http://pysvn.tigris.org/docs/pysvn_prog_ref.html

import pysvn
client = pysvn.Client()


#USING LOCAL SVN REPOS
project = 'project'

#Typical url and path
url= 'file:///c:/svnserve/'+project
path = 'c:/svncheckout/'+project

RevNumber = 0
Checkout1Update0 = 1

```

```
#check out the current version of the pysvn project
###################################################
#client.checkout(url, path)

#check out revision number of the project
##########################################
if Checkout1Update0 == 1:
    print 'Checking out revision: '+str(RevNumber)+' of '+project
    client.checkout(url,
    path,
    revision=pysvn.Revision(pysvn.opt_revision_kind.number, RevNumber),
    ignore_externals=False)
else:
#update the local revision to new revision number
#################################################
    print 'Updating '+project+' to revision: '+str(RevNumber)
    client.update(path,
    revision=pysvn.Revision(pysvn.opt_revision_kind.number, RevNumber),
    ignore_externals=False)

f=open('c:/svncheckout/'+project+'_current_rev.txt', 'w')
f.write(str(RevNumber))
f.close()
```

19-ExtractCallChangesPerRevision_withDIRSv2.py

```
'''
This first chunk here is copied from
http://pysvn.tigris.org/svn/pysvn/trunk/pysvn/WorkBench/Source/wb_main.py

It is required to avoid a locale error when running on Linux. It was not
needed just to run in Windows.
Windows, on the other hand, can't deal with two files that differ in case.
For example: 'file' and 'FILE' are two different things in UNIX but not in
Windows.

So, now we are running the code in Linux to avoid the naming problem, but
had to add the locale code...
'''

import sys
import locale
import os

def initLocale():
    # init the locale
    if sys.platform == 'win32':
        locale.setlocale( locale.LC_ALL, '' )

    else:
        if 'LC_ALL' in os.environ:
            try:
                locale.setlocale( locale.LC_ALL, os.environ['LC_ALL'] )
                return
            except locale.Error:
                pass

        language_code, encoding = locale.getdefaultlocale()
        if language_code is None:
            language_code = 'en_US'

        if encoding is None:
            encoding = 'UTF-8'
        if encoding.lower() == 'utf':
            encoding = 'UTF-8'

        try:
            # setlocale fails when params it does not understand are passed
            locale.setlocale( locale.LC_ALL, '%s.%s' % (language_code, encoding) )
        except locale.Error:
            try:
                # force a locale that will work
                locale.setlocale( locale.LC_ALL, 'en_US.UTF-8' )
            except locale.Error:
                locale.setlocale( locale.LC_ALL, 'C' )

initLocale()
print 'Info: locale set to %r' % (locale.getlocale(),)


'''
Now my old code:
*********************************************************************************************
********
0-Repository is in revision X (seed)
1-Get the changes that were made in revision X+1
2-Check the imports on the files changed in X+1, in the revision number X.
3-Update repository to X+1
4-Check the imports on the files in X+1
5-If changes were made update delta log

Delta log:
Revision #
PathAndFileName library (a)dded or (d)eleted
PathAndFileName library (a)dded or (d)eleted
PathAndFileName library (a)dded or (d)eleted
PathAndFileName library (a)dded or (d)eleted

Revision #
PathAndFileName library (a)dded or (d)eleted
'''

import MySQLdb
import os
import Extractor
import pysvn
import time
import random

# Project
project='project'

url= 'file:///c:/svnserve/'+project
path = 'c:/svncheckout/'+project

totalDelay=0
delayFactor = 0 #how many seconds to wait between requests to the server

# Open connection to database
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

# get the highest commit number
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])

StartCommit = 0

f=open('ChangesPerRevision_'+project+'-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')
start = time.localtime()
previous = time.localtime()

for CurrentCommit in range(StartCommit,TopCommit):
    ##########################################################
    # Start the analysis


    # Step 1 - Get the CHANGES that were made in revision CurrentCommit+1
    dbcursor.execute("select file from rev_files where commitNum="+str(CurrentCommit+1))
    FilesChanged = dbcursor.fetchall()

    if len(FilesChanged)>0: #skip revisions without any changes in it (see project Zope for example)

        # Step 2 - Check what state these were in in CurrentCommit
        BeforeLinks=[]
        for i in range (0, len(FilesChanged)):
            ObjName = FilesChanged[i][0]
            if os.path.isfile ( path+ObjName ): # check that it is a file
                Links = Extractor.ExtractCallsIgnoreComments(path+ObjName ,0) #Gets the calls in a file
                for j in range (0, len(Links)):
                    Links[j]=[path+ObjName,Links[j][0]] # For each link found puts in the file that is originating the call next to it
                BeforeLinks = BeforeLinks+Links
                Links=[]

            #if there is a directory in the change list and it is being deleted, then the files inside it need to be on the edit list to
            if os.path.isdir (path+ObjName):
                dbcursor.execute("select changeType from rev_files where file='"+ObjName+"' AND commitNum="+str(CurrentCommit+1))
                changeType = dbcursor.fetchall()
                if changeType[0][0].lower()=='deleted':

                    #it is a folder and it is being deleted. add all the subfiles to the BeforeLinks. Work recursively finding all files under this folder.

                    #print 'dir changed: '+str(CurrentCommit+1)+' -'+ObjName+' -'+changeType[0][0]
                    SubObjList = [path+ObjName]
                    subfinds = 0
                    for SubObj in SubObjList:
                        print 'checking in '+SubObj
                        for SubObjName in os.listdir ( SubObj ):
                            print 'item: '+SubObjName
                            if os.path.isfile ( SubObj+'/'+SubObjName ):
                                #print 'found file:'+SubObjName+' in delete dir '+ObjName
                                Links = Extractor.ExtractCallsIgnoreComments(SubObj+'/'+SubObjName ,0) #Gets the calls in a file
                                for j in range (0, len(Links)):
                                    Links[j]=[SubObj+'/'+SubObjName,Links[j][0]] # For each link found puts in the file that is originating the call next to it
                                    #print 'special found: '+Links[j][0]+' - '+Links[j][1]
                                BeforeLinks = BeforeLinks+Links
                                Links=[]
                                subfinds = subfinds+1
                            if os.path.isdir (SubObj+'/'+SubObjName):
                                if SubObjName != '.svn':
                                    print 'found subdir: '+SubObj+'/'+SubObjName +' ('+str(CurrentCommit+1)+')'
                                    SubObjList.append(SubObj+'/'+SubObjName)

        # Step 3 - Update repository to X+1
        client = pysvn.Client()

    #    client.update(path+'/trunk',
        client.update(path,
            revision=pysvn.Revision(pysvn.opt_revision_kind.number, CurrentCommit+1),
            ignore_externals=False)

        f_rev=open('c:/svncheckout/'+project+'_current_rev.txt', 'w')
        f_rev.write(str(CurrentCommit+1))
        f_rev.close()

        #print 'updated to '+str(CurrentCommit)

        # Step 4 - Check the imports on the files in X+1
        AfterLinks=[]
        for i in range (0, len(FilesChanged)):
            ObjName = FilesChanged[i][0]
            if os.path.isfile ( path+ObjName ): # check that it is a file
                    # check that it is a source code file
                Links = Extractor.ExtractCallsIgnoreComments(path+ObjName, 0) #Gets the calls in a file
                for j in range (0, len(Links)):
                    Links[j]=[path+ObjName,Links[j][0]] # Adds back the file that is originating the call
                AfterLinks = AfterLinks+Links
                Links=[]
            #else:
            #    print "didn't exist or is not file"

        # Step 5 - If changes were made update delta log

        '''
        # step 5.1 - visualize diffs
        print '###BeforeLinks'
        for i in range (0, len(BeforeLinks)):
            print BeforeLinks[i][0].rsplit('/',1)[1]+'->'+BeforeLinks[i][1]
        print '###AfterLinks'
        for i in range (0, len(AfterLinks)):
            print AfterLinks[i][0].rsplit('/',1)[1]+'->'+AfterLinks[i][1]
        '''


        # step 5.2 - compare lists
        diffPlus = AfterLinks[:]
        for item in AfterLinks:
            if item in BeforeLinks:
                diffPlus.remove(item)

        diffMinus = BeforeLinks[:]
        for item in BeforeLinks:
            if item in AfterLinks:
                diffMinus.remove(item)

        # step 5.3 - write to delta log
        # Revision# <> ADD or REM <> file <> target

        if len(diffPlus)>0:
            for i in range(0, len(diffPlus)):
                f.write(str(CurrentCommit+1)+'\tADD\t'+diffPlus[i][0]+'\t'+diffPlus[i][1]+'\n')

        if len(diffMinus)>0:
            for i in range(0, len(diffMinus)):
                f.write(str(CurrentCommit+1)+'\tREM\t'+diffMinus[i][0]+'\t'+diffMinus[i][1]+'\n')


        print 'commmit '+str(CurrentCommit+1)+' -'+str(len(FilesChanged)) +' files changed, '+str(len(diffPlus))+' links (added) and '+str(len(diffMinus))+' links (removed)'

        # Step 6 - be nice to server and wait a bit
        #delayNow = delayFactor*random.random()
        #time.sleep (delayNow)
```

```
            #totalDelay = totalDelay + delayNow

        if CurrentCommit % 250 == 0:
            current = time.localtime()
            print 'Done: '+str(CurrentCommit)+' - Last 250: '+str(time.mktime(current)-time.mktime(previous))+' - Total: '+str(time.mktime(current)-time.mktime(start))
            previous = current

    f.close()

    finish = time.localtime()

    print 'time elapsed: '+str(time.mktime(finish)-time.mktime(start))+'s, with '+str(totalDelay)+'s delay introduced'
```

23-RevisionProfiles - Files,Links Added & Removed.py

```
'''
For each revision counts how many files were changed and builds a histogram.

'''


import MySQLdb

# Project
project='project'
#project='gnumeric'
StartRev=1
EndRev=16627 # if you put a rev number larger than what exists on file the script will break.
            # in that case make sure to .close() the files and you will still get the right
            # results. The same number could also be extracted directly from the database.

#initialize variables
Path ='../3. Results/ChangesPerRevision/'+project
File ='3.UNoLibs&Reduce_ChangesPerRevision_+'project+'.txt'
lineNum=0
AddCount = 0
RemCount = 0

print 'Starting work on: '+project

CallChangesFile=open(Path+'/'+File, 'r')
CallChangesList = CallChangesFile.readlines()

result=open('RevisionProfiles_'+project+'_output.txt', 'w')
result.write('Rev\t#Files\tAdd\tRem\n')


# Open connection to database
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

for rev in range(StartRev,EndRev):
    # get all the filenames and insert them in a dictionary.
    # dictionary includes the index number that will be used as the table index for that file
    dbcursor.execute("select count(file) from rev_files where commitNum="+str(rev))
    NumberOfFilesEdited = dbcursor.fetchall()

    NumberOfFilesEdited[0][0]

    if lineNum <len(CallChangesList):
        while int(CallChangesList[lineNum].split('\t')[0]) == rev:
            if CallChangesList[lineNum].split('\t')[1]=='ADD':
                AddCount = AddCount +1
            if CallChangesList[lineNum].split('\t')[1]=='REM':
                RemCount = RemCount +1
            lineNum = lineNum+1
            if lineNum >=len(CallChangesList):
                break


    result.write(str(rev)+'\t'+str(NumberOfFilesEdited[0][0])+'\t'+str(AddCount)+'\t'+str(RemC
ount)+'\n')
    AddCount = 0
    RemCount = 0

CallChangesFile.close()
result.close()

print 'Done'
```

24-CallGraph&RevisionRecord-Fit3-nx.py

```python
'''
For each revision see how it compares with the function call graph of
the program up to that point.

Outputs a text file with:
- the count of links in a specific revision and links matched by the function
call

- also sees if the matching link was just created by the revision.
- also keeps track of unmatched links to count how many times they occur again
or if they are created later

Also outputs a edit record graph by revision with the following format:
Revision# <> ADD <> sourcefile <> targetfile

Note: there are only adds in the edit record (links can't be removed).
The same link may have multiple ADD if the source and target were edited
together multiple times

Note 2: I'm using an adapted version of the NetStateOnRevision function, but not
running the function because that one reads all the files up to the Rev # before
returning the results, so it would redundantly read the start of file many times
over.

EDIT ON AUG 14 2009
Major rewrite of the code to use networkX library instead
***********************************************************************************************************'''

import time
import MySQLdb
import Path
import networkx as nx

# Project
project='chandler'
#project='zope'

# initialize vars
CallGraph = []
EditGraph = []
CallGraphLinePointer = 0
rootpath = '../svncheckout/'+project                    # pick line depending on if script 19-ExtractCallChangesPerRevision.py
#rootpath = 'c:/svncheckout/'+project                    # was run on Windows or Ubuntu. Don't need to include the /trunk/ portion
#rootpath = '/home/joao/Documents/svncheckout/'+project  # as it is already present in the database and will be concatenated below
                                                         # check the file ReduceLog_project to see which one applies
progress = 250 #show progress every X steps
previous = time.localtime()
start = time.localtime()

useDebug = 0
useEditGraphByRev = 0


def updateCGtoRev():
    global CallGraphLinePointer

    if CallGraphLinePointer < len(CallGraphList):
        while int(CallGraphList[CallGraphLinePointer].split('\t')[0]) <= revNum: #if the line points to an update that belongs in the future, then it skips
            if CallGraphList[CallGraphLinePointer].split('\t')[1] == 'ADD':
                Source = CallGraphList[CallGraphLinePointer].split('\t')[2]
                Target = CallGraphList[CallGraphLinePointer].split('\t')[3].replace('\n','')

                if [Source,Target] not in CallGraph:
                    #add to list
                    CallGraph.append([Source,Target])

            if CallGraphList[CallGraphLinePointer].split('\t')[1] == 'REM':
                if [CallGraphList[CallGraphLinePointer].split('\t')[2],CallGraphList[CallGraphLinePointer].split('\t')[3].replace('\n','')] in CallGraph:
                    #remove from list
                    CallGraph.remove([CallGraphList[CallGraphLinePointer].split('\t')[2],CallGraphList[CallGraphLinePointer].split('\t')[3].replace('\n','')])

            CallGraphLinePointer += 1 # update pointer for next line
            if CallGraphLinePointer == len(CallGraphList): #TRIAL - tests if we have reached the end of the file
                break
    return



# Step 0 - Setup
# connect to db
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

#get how many revisions we have to process
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])
#TopCommit = 24 # overwritten changed for debugging
StartCommit = 0


#open files
CallGraphFile = open ('../3. Results/ChangesPerRevision/'+project+'/3.UNoLibs&Reduce_ChangesPerRevision_'+project+'.txt', 'r')
CallGraphList = CallGraphFile.readlines()
CallGraphFile.close()

CallAndRev     = open ('../3. Results/24-H2-CallAndRevFit/'+project+'/bCallAndRev_nx'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')
CallAndRev.write('Note: Links is the number of pairs. Directed links would be twice that number.\n')

SubGraphsByRev = open ('../3. Results/24-H2-CallAndRevFit/'+project+'/bSubGraphsByRev'+project+'_nx'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')

if useDebug ==1:
    debugLog       = open ('../3. Results/24-H2-CallAndRevFit/'+project+'/debugLog_nx'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')
if useEditGraphByRev ==1:
    EditGraphByRev = open ('../3. Results/24-H2-CallAndRevFit/'+project+'/EditGraphByRev_'+project+'_nx'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')




# Start Here

for revNum in range (StartCommit, TopCommit+1):

    if revNum % progress == 0:
        current = time.localtime()
        print str(revNum)+' - '+str(time.mktime(current)-time.mktime(previous))
        previous = current

    #print '############## '+str(revNum)+' ##############'

    #get revision information from db
    dbcursor.execute("select file from rev_files where commitNum="+str(revNum))
    editRecord = dbcursor.fetchall()

    #we only care about revisions that edit more than one file
    if len(editRecord) > 1:
        #create list of links in rev
        editRecordPairs = []
        linksInRev = 0
        created=0
        reuse=0
        numOfCodeFilesInRev=0
        subgraphs_nx=nx.Graph()
        subgraphs_nx.clear()
        editNodeList=[]

        for i in range (0, len(editRecord)):
            for j in range (i+1, len(editRecord)):
                # we only care about python files connected to other python files (endswith .py)
                # and that are part of the chandler code (startswith /trunk/chandler) and not
                # from other sections of code (internal, external and hardhat)
                if editRecord[i][0].startswith('/trunk/'+project) and
```

```
editRecord[i][0].endswith('.py') and\
                   editRecord[j][0].startswith('/trunk/'+project) and
editRecord[j][0].endswith('.py'):
                    pairA = Path.reducedot(rootpath+editRecord[i][0],project,1,revNum)
                    pairB = Path.reducedot(rootpath+editRecord[j][0],project,1,revNum)
                    editRecordPairs.append([pairA, pairB, 0]) #last item (0) is a flag to be
used later on
                    linksInRev = linksInRev +1
                    if pairA not in editNodeList:
                        editNodeList.append(pairA)
                    if pairB not in editNodeList:
                        editNodeList.append(pairB)

                    #write in fileB- rev#, ADD, file i, file j
                    if useEditGraphByRev == 1:
EditGraphByRev.write(str(revNum)+'\tADD\t'+editRecord[i][0]+'\t'+editRecord[j][0]+'\n')
                    #remeber that these links are NOT oriented. although we build the matrix
of pairs, we should take care that A-B is as valid as B-A.

        #store this rev in the complete EditGraph - all changes that have been made between
files over all previous edits.
        '''
        for item in editRecordPairs:
            if item not in EditGraph:
                EditGraph.append(item)
            #else
            #    EditGraph increment the count of occurrences
        '''

        #compare with previous rev call graph so we can see if the links were already there or
if they are new
        revNum = revNum-1
        updateCGtoRev()
        revNum = revNum+1

        for item in editRecordPairs:
            index = editRecordPairs.index(item)

            if [item[0],item[1]] in CallGraph:
                #set the flag "was already there" to one
                #print "01 found old one: "+item[0]+' - '+item[1]
                editRecordPairs[index]=[editRecordPairs[index][0],editRecordPairs[index][1],1]

            if [item[1],item[0]] in CallGraph:
                #set the flag "was already there" to one
                editRecordPairs[index]=[editRecordPairs[index][0],editRecordPairs[index][1],1]
                #print "10 found old one: "+item[1]+' - '+item[0]

            if useDebug ==1:
                debugLog.write(str(revNum)+' > '+item[0]+'  <>  '+item[1]+'\n')


        # update call graph to revision
        updateCGtoRev()


        #compare with current rev call graph
        for item in editRecordPairs:

            if [item[0],item[1]] in CallGraph:
                index = editRecordPairs.index(item)
                if editRecordPairs[index][2]==0:
                    #print "found new one: "+item[0]+' - '+item[1]
                    created = created+1
                    #subgraph_tracker()
                    subgraphs_nx.add_edge(item[0],item[1])
                if editRecordPairs[index][2]==1:
                    #print "re-used match: "+item[0]+' - '+item[1]
                    reuse = reuse+1
                    #subgraph_tracker()
                    subgraphs_nx.add_edge(item[0],item[1])

            if [item[1],item[0]] in CallGraph:
                index = editRecordPairs.index(item)
                if editRecordPairs[index][2]==0:
                    #print "found new one: "+item[0]+' - '+item[1]
                    created = created+1
                    #subgraph_tracker()
                    subgraphs_nx.add_edge(item[0],item[1])
                if editRecordPairs[index][2]==1:
                    #print "re-used match: "+item[0]+' - '+item[1]
                    reuse = reuse+1
                    #subgraph_tracker()
                    subgraphs_nx.add_edge(item[0],item[1])

        #print_subgraphs()
        if nx.number_connected_components(subgraphs_nx)>0:
            SubGraphsByRev.write('############## '+str(revNum)+' ##############\n')
            for i in range(0,nx.number_connected_components(subgraphs_nx)):
                #SubGraphsByRev.write(str(subgraphs[i])+'\n')
                SubGraphsByRev.write(str(nx.connected_components(subgraphs_nx)[i])+'\n')
                #print nx.connected_components(subgraphs_nx)[i]

        # how many files edited in this rev
        for i in range (0, len(editRecord)):
            if editRecord[i][0].endswith('.py'):
                numOfCodeFilesInRev = numOfCodeFilesInRev+1

        # how many files ended up isolated (no call graph connection connecting to other files edited in this rev)
        isolates=0
        for i in editNodeList:
            if i not in subgraphs_nx:
                SubGraphsByRev.write('[\''+str(i)+'\']\n')
                isolates += 1


        if linksInRev>0:
            output = 'Rev# '+str(revNum)+'\tFiles:Links: '+str(numOfCodeFilesInRev)+':'+str(linksInRev)\
                             +'\tLinks Reused: '+str(reuse)+'\tLinks Created: '+str(created)\
                             +'\tSubGraphs: '+str(nx.number_connected_components(subgraphs_nx))+'\tIsolates: '+str(isolates)
            CallAndRev.write(output+'\n')
            print output
            #NOTE: nx.number_connected_components(subgraphs_nx) only gives us the right number of subgraphs with links
            # because we only added edges to subgraphs_nx. If we had added nodes, isolated nodes would also count as
            # components!
            # That's why we later on check for nodes that did not get included in the subgraphs_nx and from there we
            # count the number of isolates.
            # Not the best way to do it but for now it works. Elegant solution requires a re-write here.

#Clean up and finish

CallAndRev.close()
SubGraphsByRev.close()
if useDebug ==1:
    debugLog.close()
if useEditGraphByRev == 1:
    EditGraphByRev.close()


print 'time elapsed: '+str(time.mktime(time.localtime())-time.mktime(start))
```

26.5-NoLibs&Reduce.py

```python
'''
Goes through a project's ChangesPerRevision file and removes all the entries related to
global or added external libraries.

Reduces the path outputs to a single name. Avoids duplicates by appending -JC

Outputs to NoLibs&Reduce_ChangesPerRevision file.
'''

import Path

project = 'project'

clearedLink =0
keptLink = 0

#load global module list
listaGlobal=[]
f=open('PythonGlobalModuleIndex.txt', 'r')
fileList = f.readlines()
for fileLine in fileList:
    fileLine = fileLine.replace ( "\n", "" )
    listaGlobal.append(fileLine)
f.close()

#load libraries known to be used by the project
f=open('ExternalLibs-'+project+'.txt', 'r')
fileList = f.readlines()
for fileLine in fileList:
    fileLine = fileLine.replace ( "\n", "" )
    if len(fileLine)>0 and fileLine[0] is not '#': #allows us to put comments in file (references to sources, for example)
        listaGlobal.append(fileLine)
f.close()

#load the full change log and create the new target file
f=open('../3. Results/ChangesPerRevision/'+project+'/2.ChangesPerRevision_'+project+'.txt','r')
fileList = f.readlines()
t=open('../3. Results/ChangesPerRevision/'+project+'/3.NoLibs&Reduce_ChangesPerRevision_'+project+'.txt','w')

#check line by line if the target link is part of the list
for line in fileList:
    linesplit = line.split('\t')
    if linesplit[3].replace('\n','').split('.')[0].lower() not in listaGlobal:
        if linesplit[3].replace('\n','').split('.')[-1].lower() not in listaGlobal:
            #reduce the file path
            linesplit[2]=Path.reducedot(linesplit[2],project,1,linesplit[0])

            #reduce the target name
            #linesplit[3] = linesplit[3].split('.')[-1] #if we wanted to only keep the last name

            # TRANSFORM TARGET INTO ITS EQUIVALENT SOURCE NAME
            # if target has no dots, then it is in the same folder as the source file
            # so copy the path from the source onto this one
            if linesplit[3].find('.') == -1:
                linesplit[3] = linesplit[2].rsplit('.',1)[0]+'.'+linesplit[3]

            '''#This section only for Chandler
            else: #if it has dots, where does it refer to?

                knownLevelOne = ['application', 'repository', 'parcels', 'tools', 'model', 'crypto', 'i18n']

                if linesplit[3].split('.')[0].lower() in knownLevelOne:
                    linesplit[3] = 'chandler.'+linesplit[3].lower()
                if linesplit[3].split('.')[0].lower() == 'osaf':
                    linesplit[3] = 'chandler.parcels.'+linesplit[3].lower()
                if linesplit[3].split('.')[0].lower() == 'util':
                    linesplit[3] = 'chandler.model.'+linesplit[3].lower()
                if linesplit[3].split('.')[0].lower() == 'chandlerdb':
                    linesplit[3] = 'internal.'+linesplit[3].lower()

                if linesplit[3].split('.')[0].lower() == 'p2p':
                    linesplit[3] = 'chandler.projects.chandler-p2pplugin.'+linesplit[3].lower()
                if linesplit[3].split('.')[0].lower() == 'debug':
                    linesplit[3] = 'chandler.projects.chandler-debugplugin.'+linesplit[3].lower()
                if linesplit[3].split('.')[0].lower() == 'feeds':
                    linesplit[3] = 'chandler.projects.chandler-feedsplugin.'+linesplit[3].lower()
                if linesplit[3].split('.')[0].lower() == 'gdata':
                    linesplit[3] = 'chandler.projects.chandler-gdataplugin.'+linesplit[3].lower()
            '''

            linesplit[3] = linesplit[3].lower()

            #write to the file
            for i in range (0, len(linesplit)-1):
                t.write(linesplit[i]+'\t')
            t.write(linesplit[-1])#which is already carrying '\n' in it so I don't need to put one

            keptLink = keptLink +1
        else:
            clearedLink = clearedLink+1
    else:
        clearedLink = clearedLink+1


t.close()
print 'Kept: '+str(keptLink)+' Removed: '+str(clearedLink)
```

28-WindowedFit.py

```
'''
Analyses the CallAndRev file and plots values in time windows.

'''

import MySQLdb
import time

#USER VARS
project='twisted'
WindowSize = 120
WindowSlide = 120
StartCommit = 0
TopCommit = 0 # 0 for entire project.

#SCRIPT VARS
CallAndRevsFit = []
RevsInWindow = []
LinePointer=1
rolling_sum = 0
WindowEnd = WindowSize

# ???
WindowFit = open('../3. Results/28-WindowFit/'+project+'/WindowFit_'+project+'-'+str(WindowSiz
e)+'-'+str(WindowSlide)+'--'+str(StartCommit)+'-'+str(TopCommit)+'.txt','w')
# Records the edits whose graph finds correspondence on the structure
WindowPerfects = open('../3. Results/28-WindowFit/'+project+'/WindowPerfects_'+project+'-'+str
(WindowSize)+'-'+str(WindowSlide)+'--'+str(StartCommit)+'-'+str(TopCommit)+'.txt','w')
# Records the edits whose graph does not find correspondence on the structure
WindowImperfects = open('../3. Results/28-WindowFit/'+project+'/WindowImperfects_'+project+'-'
+str(WindowSize)+'-'+str(WindowSlide)+'--'+str(StartCommit)+'-'+str(TopCommit)+'.txt','w')


# connect to db
db = MySQLdb.connect(host="localhost",
                     user="username",
                     passwd="password",
                     db=project)
dbcursor = db.cursor()

#get how many revisions we have to process
if TopCommit == 0:
    dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
    TopCommit = dbcursor.fetchall()
    TopCommit = int(TopCommit[0][0])
dbcursor.execute("select devdays from rev_commit where commitNum="+str(TopCommit))
WindowFinish = dbcursor.fetchall()
WindowFinish = WindowFinish[0][0]


#LOAD UP DATA
#Call and Rev fit results
f = open('../3.
Results/24-H2-CallAndRevFit/'+project+'/CallAndRev_nx0-'+str(TopCommit)+'.txt','r')
for element in f.readlines():
    CallAndRevsFit.append(element.lstrip('Rev# '))
f.close()

''' SLIDING WINDOW'''

while WindowEnd <= WindowFinish+ WindowSlide:
    rolling_sum  = 0
    perfects_sum = 0
    imperfects_sum = 0
    RevsInWindow = []
    #Get Revs that fit in the window
    dbcursor.execute('select commitNum from rev_commit where devdays >='+str(WindowEnd-
WindowSize)+' AND devdays <'+str(WindowEnd)+' order by devdays')
    result = dbcursor.fetchall()
    for element in range(0, len(result)):
        RevsInWindow.append(int(result[element][0]))

    print '\n###########'+str(max(RevsInWindow))

    # Note: the revs are not ordered by date!
    # we souldn't just use the max of RevsInWindow to determine the end of our interval.
    # we have to look at the specific revNums that are part of that time interval.
    # some revNums might be < than the max that is in the window and those revNums may belong
```

```
to another interval!

    #Add values inside the window
    while 1:
        #If we have past the end of the file
        if LinePointer>=len(CallAndRevsFit):
            print 'done up to '+str(max(RevsInWindow))+' line-'+str(LinePointer)
            print 'Rolling Sum: '+str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(rolling_sum)
            print 'Perfects: '+str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(perfects_sum)
            print 'Imperfects: '+str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(imperfects_sum)
            WindowFit.write(str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(rolling_sum)+'\n')
            WindowPerfects.write(str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(perfects_sum)+'\n')
            WindowImperfects.write(str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(imperfects_sum)+'\n')
            break
        #Skip values before the window
        if int(CallAndRevsFit[LinePointer].split('\t')[0])< min(RevsInWindow):
            LinePointer=LinePointer+1 #Update just the position of the pointer
        else:
            #print CallAndRevsFit[LinePointer].split('\t')[0]
            if int(CallAndRevsFit[LinePointer].split('\t')[0])<= max(RevsInWindow):
                subgraphs = int(CallAndRevsFit[LinePointer].split('\t')[4].split(': ')[1])
                isolates  = int(CallAndRevsFit[LinePointer].split('\t')[5].split(': ')[1])
                #print str(subgraphs)+' - '+str(isolates)
                if subgraphs + isolates > 1:
                    rolling_sum = rolling_sum + subgraphs + isolates
                    imperfects_sum += 1
                    #print str(LinePointer)+' '+str(imperfects_sum)
                else:
                    perfects_sum += 1
                LinePointer=LinePointer+1 #Update the position of the pointer
            #Skip values after the window
            else:
                print 'done up to '+str(max(RevsInWindow))+' line-'+str(LinePointer)
                print 'Rolling Sum: '+str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(rolling_sum)
                print 'Perfects: '+str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(perfects_sum)
                print 'Imperfects: '+str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(imperfects_sum)
                WindowFit.write(str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(rolling_sum)+'\n')
                WindowPerfects.write(str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(perfects_sum)+'\n')
                WindowImperfects.write(str(WindowEnd-WindowSize)+' - '+str(WindowEnd)+': '+str(imperfects_sum)+'\n')
                break

    #Update the WindowEnd position
    WindowEnd = WindowEnd + WindowSlide

WindowFit.close()
WindowPerfects.close()
WindowImperfects.close()
```

29-Insert_DevDays_in_DB.py

```
'''
Converts the 'date' timestamp in the database into a numeral (number of days in development).

1- gets the project start date
2- gets how many revisions there are
3- calculates the difference betwen a revision's date and the project start date
4- updates the database

NOTE: table schema in DB has to be updated before this can be run:
    ALTER TABLE rev_commit ADD devdays DOUBLE;

DATE MANIPULATION:
    Database has dates stored as: '7:49:18 AM, Tuesday, November 18, 2008'
    temp = '6:50:53 PM, Sunday, November 02, 2008'
    pytemp = time.strptime(temp, '%I:%M:%S %p, %A, %B %d, %Y')
    #result is in the format of (year, mon, mday, hour, min, sec, wday, yday, isdst)

'''

import MySQLdb
import time

project='projectname'

# connect to db
db = MySQLdb.connect(host="localhost",
                     user="username",
                     passwd="password",
                     db=project)
dbcursor = db.cursor()

#get how many revisions we have to process
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])
#TopCommit = 25 # overwritten changed for debugging
StartCommit = 1

# get first project date
dbcursor.execute("select date from rev_commit order by commitNum limit 1")
StartDate = dbcursor.fetchall()
StartDate = time.strptime( StartDate[0][0], '%I:%M:%S %p, %A, %B %d, %Y')

for revNum in range (StartCommit, TopCommit+1):
    dbcursor.execute("select date from rev_commit where commitNum ="+str(revNum))
    revDate = dbcursor.fetchall()
    if len(revDate)>0:
        diff = time.mktime(time.strptime( revDate[0][0], '%I:%M:%S %p, %A, %B %d, %Y'))-time.mktime(StartDate)
        devdays = diff/60/60/24
        #print "UPDATE rev_commit SET devdays="+str(devdays)+" WHERE commitNum="+str(revNum)
        dbcursor.execute("UPDATE rev_commit SET devdays="+str(devdays)+" WHERE commitNum="+str(revNum))

print 'Done.'
```

30-H4-EditBecomesStruct_v2.py

```python
'''
Checks if graph from a specific revision becomes structure in the last revision

v2- updateCGtoRev now uses networkx
************************************************************************************************************
'''

import time
import MySQLdb
import Path
import networkx as nx

# Project
project='project'

# initialize vars
CallGraph = []
EditGraph = []
CallGraphLinePointer = 0

#rootpath = '../svncheckout/'+project                  #chandler    # pick line depending on if script 19-ExtractCallChangesPerRevision.py
#rootpath = 'c:/svncheckout/'+project                   #trac        # was run on Windows or Ubuntu. Don't need to include the /trunk/ portion
rootpath = '/home/joao/Documents/svncheckout/'+project #twisted/zope    # as it is already present in the database and will be concatenated below
                                                                    # check the file ReduceLog_project to see which one applies

progress = 250 #show progress every X steps
previous = time.localtime()
start = time.localtime()

useDebug = 0
useEditGraphByRev = 0

#convert CallGraph edge list to networkX
callGraphX=nx.Graph()
callGraphX.clear()

def updateCGtoRev_nx():
    global CallGraphLinePointer

    if CallGraphLinePointer < len(CallGraphList):
        while int(CallGraphList[CallGraphLinePointer].split('\t')[0]) <= revNum: #if the line points to an update that belongs in the future, then it skips
            if CallGraphList[CallGraphLinePointer].split('\t')[1] == 'ADD':
                Source = CallGraphList[CallGraphLinePointer].split('\t')[2]
                Target = CallGraphList[CallGraphLinePointer].split('\t')[3].replace('\n','')

                if [Source,Target] not in CallGraph:
                    #add to list
                    callGraphX.add_edge(Source,Target)

            if CallGraphList[CallGraphLinePointer].split('\t')[1] == 'REM':
                if [CallGraphList[CallGraphLinePointer].split('\t')[2],CallGraphList[CallGraphLinePointer].split('\t')[3].replace('\n','')] in CallGraph:
                    #remove from list
                    callGraphX.remove_edge(CallGraphList[CallGraphLinePointer].split('\t')[2], CallGraphList[CallGraphLinePointer].split('\t')[3].replace('\n',''))

            CallGraphLinePointer += 1 # update pointer for next line
            if CallGraphLinePointer == len(CallGraphList): #TRIAL - tests if we have reached the end of the file
                break
    return



# Step 0 - Setup
# connect to db
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

#get how many revisions we have to process
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])
StartCommit = 0


#open files
CallGraphFile = open ('../3.
Results/ChangesPerRevision/'+project+'/3.NoLibs&Reduce_ChangesPerRevision_'+project+'.txt',
'r')
CallGraphList = CallGraphFile.readlines()
CallGraphFile.close()

CallAndRevFit = open ('../3. Results/24-H2-CallAndRevFit/'+project+'/CallAndRev_nx'+str(StartC
ommit)+'-'+str(TopCommit)+'.txt', 'r')
CallAndRevFit_lines = CallAndRevFit.readlines()
CallAndRevFit.close()

SubGraphsByRev = open ('../3. Results/24-H2-CallAndRevFit/'+project+'/SubGraphsByRev_'+project
+'_nx'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'r')
subgraphs_lines = SubGraphsByRev.readlines()
SubGraphsByRev.close()

EditBecomesStruct = open ('../3. Results/30-H4-EditBecomesStruct/'+project+'/EditBecomesStruct
_'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')
BeforeAndAfterRatio = open ('../3. Results/30-H4-EditBecomesStruct/'+project+'/BeforeAndAfterR
atio_'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')


###### Start Here

#TopCommit = 2000 # overwritten changed for debugging


# Get the network state of the product at the final revision
revNum = TopCommit
updateCGtoRev_nx()


# initialize variables
nodeList =[]
currentRev =0

subgraphX=nx.Graph()
subgraphX.clear()

#editRecordPairs =[]
missingNodes = []
missing =0
isolates =0
perfectFitsList =[]
isolatesList =[]
missingList =[]
whatType =[]
count=0
LinksRequiredForConnected = {0:1}

#extract the values from CallAndRevFit_lines
for line in CallAndRevFit_lines:
    line = line.split('\t')
    if len(line)>1:
        key = int(line[0].replace('Rev# ',''))
        required = (int(line[4].replace('SubGraphs: ',''))+     #these subgraphs do not
include the isolates
                    int(line[5].replace('Isolates: ','').replace('\n',''))+
                   -1)
        LinksRequiredForConnected[key]=required

### ########################################################################
### for each revision, load the subgraphs and compare with the final structure

for line in subgraphs_lines:
    if line.startswith('###############'):
        #print str(currentRev)
        #close and calculate previous Rev match
        if LinksRequiredForConnected[int(currentRev)] > 0:
            #print str(currentRev)+' '+str(LinksRequiredForConnected[int(currentRev)])

            ### check if the node connections are in the CallGraph, to see if any have been
deleted
            for node in nodeList:
                if not callGraphX.has_node(node):
                    missing = missing +1
                    if node not in missingNodes:
                        missingNodes.append(node)
                        #print str(currentRev)+': deletedOrDisconnected '+node
                #else:
                    #print str(currentRev)+': found '+node

            ### select the subsection of the CallGraph with the nodes in this edit
            subgraphX = nx.subgraph(callGraphX, nodeList)

            ### find isolates in the subgraph
            for i in nx.nodes(subgraphX):
                if nx.degree(subgraphX,i) == 0:
                    isolates = isolates +1


            ### Write output
            EditBecomesStruct.write('Rev#: '+str(currentRev)\
            #print ('Rev#: '+str(currentRev)\
                                    +'\t#OriginalNodes: '+str(len(nodeList))\
                                    +'\t#LinksRequired4Connected: 
'+str(LinksRequiredForConnected[int(currentRev)])\
                                    +'\t#FinalComponents: 
'+str(nx.number_connected_components(subgraphX))\
                                    +'\t#FinalIsolates: '+str(isolates)\
                                    +'\t#deletedOrDisconnected: '+str(missing)\
                                    +'\t#Finalnodeinsubgraph: 
'+str(subgraphX.number_of_nodes())\
                                    +'\n')
            if len(nodeList)>0:
                BeforeAndAfterRatio.write('Rev#: '+str(currentRev)\
                #print ('Rev#: '+str(currentRev)\
                                          +'\tOriginal: 
'+str(LinksRequiredForConnected[int(currentRev)]/float(len(nodeList)-1))\
                                          +'\tFinal: '+str((nx.number_connected_components(sub
graphX)+missing-1)/float(subgraphX.number_of_nodes()+missing-1))\
                                          +'\tDisappeared nodes: 
'+str(len(nodeList)-subgraphX.number_of_nodes())+'/'+str(len(nodeList))
                                          +'\n')

            if nx.number_connected_components(subgraphX)==1 and isolates==0: #all fit 
(ignoring deleted)
                perfectFitsList.append(currentRev)
            elif nx.nodes(subgraphX)==isolates: #all isolates
                isolatesList.append(currentRev)
            elif nx.nodes(subgraphX)==missing: #all deleted
                missingList.append(currentRev)
            else:
                whatType.append(currentRev)


        #prepare new rev
        line = line.replace('##############','')
        currentRev = line.strip()
        nodeList =[]
        subgraphX.clear()
        missing =0
        isolates =0
        count = count+1

        # end if we have reached TopCommit
        if int(currentRev) > TopCommit:
            break



    else:
        nodeList = nodeList+eval(line)


EditBecomesStruct.close()
BeforeAndAfterRatio.close()
print 'time elapsed: '+str(time.mktime(time.localtime())-time.mktime(start))
```

32-WhenFilesChangedInProject.py

```python
'''
Log of files changed in a project that are part of the code.
Uses the Path.reducedot function to remove the external files that are included
in just the simple file change dump from the database
'''

import MySQLdb
import Path

project = 'project'

# Step 0 - Setup
# connect to db
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

#get how many revisions we have to process
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])
#TopCommit = 30 # overwritten changed for debugging
StartCommit = 0

bad = 0
RevsWithMultipleFilesChanged = []


for revNum in range (StartCommit, TopCommit):
    dbcursor.execute("select file from rev_files where commitNum="+str(revNum))
    files = dbcursor.fetchall()

    filechangedinRev = 0

    for item in files:
        filepath = Path.reducedot(item[0],project,0,revNum)
        #print str(revNum)+' - '+item[0]+' - '+filepath
        #check if this path is of interest (part of main code and python file)

        if filepath.startswith('.trunk.'+project) and item[0].endswith('.py'):  ## THIS LINE TO WORK WITH CHANDLER
        #if (filepath.startswith('.trunk.svntrac') or filepath.startswith('.trunk.trac')) and item[0].endswith('.py'):  ## THIS LINE TO WORK WITH TRAC
        #if filepath.startswith('.zope.trunk.') and item[0].endswith('.py'):  ## THIS LINE TO WORK WITH ZOPE

            filechangedinRev += 1
            #print 'got '+str(filechangedinRev)
            if filechangedinRev >1 and revNum not in RevsWithMultipleFilesChanged:
                RevsWithMultipleFilesChanged.append(revNum)
                #print str(revNum)+' is a multifile edit'
                break

multifiles = open('../3. Results/28-WindowFit/'+project+'/Helper-EditsWithMultipleFiles.txt','w')
for i in RevsWithMultipleFilesChanged:
    multifiles.write(str(i)+'\n')
multifiles.close()

print 'done multifiles. starting struct changes'

'''PART 2
Which edits change the structure
'''
RevsWithStructChanges = []

changesPerRev = open('../3. Results/ChangesPerRevision/'+project+'/3.NoLibs&Reduce_ChangesPerRevision_'+project+'.txt','r') #If on Windows
#changesPerRev = open('../3. Results/ChangesPerRevision/'+project+'/3.UNoLibs&Reduce_ChangesPerRevision_'+project+'.txt','r') #If on Ubuntu (the change is just a U in the filename)
lines = changesPerRev.readlines()

for line in lines:
    revInLine = line.split('\t')[0]
    if revInLine not in RevsWithStructChanges:
        RevsWithStructChanges.append(revInLine)

structchange = open('../3. Results/28-WindowFit/'+project+'/Helper-EditsWithStructChanges.txt','w')
```

```
    for i in RevsWithStructChanges:
        structchange.write(str(i)+'\n')
    structchange.close()
```

33-H5-BetweenessCentralityPerRevision.py

```python
'''
For each revision calculate the betweeness centrality:
- save the names of the 50 nodes with highest betweeness centrality
- save the values, before and after of the nodes edited in this revision

'''
import networkx as nx
import MySQLdb
import time
import operator
import Path

# Project
project='project'

progress = 250 #show progress every X steps


#########################################
def updateCGtoRev():
    global CallGraphLinePointer

    if CallGraphLinePointer < len(CallGraphList):
        while int(CallGraphList[CallGraphLinePointer].split('\t')[0]) <= revNum: #if the line points to an update that belongs in the future, then it skips
            if CallGraphList[CallGraphLinePointer].split('\t')[1] == 'ADD':
                Source = CallGraphList[CallGraphLinePointer].split('\t')[2]
                Target = CallGraphList[CallGraphLinePointer].split('\t')[3].replace('\n','')

                if [Source,Target] not in CallGraph:
                    #add to list
                    CallGraph.append([Source,Target])

            if CallGraphList[CallGraphLinePointer].split('\t')[1] == 'REM':
                if [CallGraphList[CallGraphLinePointer].split('\t')[2],CallGraphList[CallGraphLinePointer].split('\t')[3].replace('\n','')] in CallGraph:
                    #remove from list
                    CallGraph.remove([CallGraphList[CallGraphLinePointer].split('\t')[2],CallGraphList[CallGraphLinePointer].split('\t')[3].replace('\n','')])

            CallGraphLinePointer += 1 # update pointer for next line
            if CallGraphLinePointer == len(CallGraphList): #TRIAL - tests if we have reached the end of the file
                break
    return

#########################################
# initialize vars
previous = time.localtime()
start = time.localtime()
CallGraph = []
CallGraph_nx = nx.Graph()
CallGraphLinePointer = 0
#rootpath = '../svncheckout/'+project #if reduce was done in Ubuntu
#rootpath = '/home/joao/Documents/svncheckout/'+project #if reduce was done in Ubuntu
rootpath = 'c:/svncheckout/'+project #if reduce was done in Windows

# Step 0 - Setup
# connect to db
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

#get how many revisions we have to process
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])
#TopCommit = 70000 # overwritten changed for debugging
StartCommit = 1

#open files
CallGraphFile = open ('../3. Results/ChangesPerRevision/'+project+'/3.NoLibs&Reduce_ChangesPerRevision_'+project+'.txt', 'r')
CallGraphList = CallGraphFile.readlines()
CallGraphFile.close()

Results = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/BCofNodesByEdit-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')
```

```

    # Start Here
    notFound = 0

    for revNum in range (StartCommit, TopCommit+1):

        if revNum % progress == 0:
            current = time.localtime()
            print str(revNum)+' - '+str(time.mktime(current)-time.mktime(previous))
            previous = current

        #get files edited in this revision from db
        dbcursor.execute("select file from rev_files where commitNum="+str(revNum))
        editRecord = dbcursor.fetchall()

        if len(editRecord)>0:

            print revNum
            nodesInEdit = []
            updateCGtoRev()

            #convert to nx
            CallGraph_nx.clear()
            for item in CallGraph:
                CallGraph_nx.add_edge(item[0],item[1])

            #calculate betweeness centrality of nodes
            bc = nx.betweenness_centrality(CallGraph_nx)

            #get top 50 nodes
            if len(bc)>1:
                sorted_bc = sorted(bc.items(), key=operator.itemgetter(1), reverse=True)
                #create a list without the values (useful to find position of nodes later in the code)
                rank_bc = [i[0] for i in sorted_bc]

                #crop just for the top 50
                if len(bc) >50:
                    top50=[]
                    for item in range(0,50):
                        top50.append(sorted_bc[item])


            #get value for nodes in current revision

            #convert name to type in graph
            for item in range(0, len(editRecord)):
                #typical IF for most projects
                if editRecord[item][0].startswith('/trunk/'+project) and editRecord[item][0].endswith('.py'):
                    nodesInEdit.append(Path.reducedot(rootpath+editRecord[item][0],project,1,revNum))

                #catering to the specific name that project trac had in its first 47 revisions
                #if editRecord[item][0].startswith('/trunk/svn'+project) and editRecord[item][0].endswith('.py'):
                #    nodesInEdit.append(Path.reducedot(rootpath+editRecord[item][0],project,1,revNum))

                #catering to Zope path style
                #if editRecord[item][0].startswith('/Zope/trunk/') and editRecord[item][0].endswith('.py'):
                #    nodesInEdit.append(Path.reducedot(rootpath+editRecord[item][0],project,1,revNum))

            #get betweeness centrality value for each node in this revision
            for nodename in nodesInEdit:
                #print nodename
                try:
                    bc[nodename]
                except KeyError:
                    #print str(revNum)+': '+nodename + ' node not found'
                    notFound += 1
                else:
                    #OUTPUT
                    #write a file with revNum and for each node: nodename, rank order number in that rev, node centrality measure
                    Results.write( str(revNum)+'\t'+nodename +'\t' +str(rank_bc.index(nodename)+1)+'/'+str(len(rank_bc)) +'\t' +str(bc[nodename])+'\n')
    Results.close()
```

34-H5-PlotResults & HeatMap (AVG)_devdays_wide.py

```
'''
For each revision calculate the betweeness centrality:
- save the names of the 50 nodes with highest betweeness centrality
- save the values, before and after of the nodes edited in this revision

'''
import MySQLdb
import time
import heatmap

# Project
project='project'


#########################################
# initialize vars
start = time.localtime()

# Step 0 - Setup
# connect to db
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

#get how many revisions we have to process
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])
#TopCommit = 16627 #last commit for chandler
#TopCommit = 5000 # overwritten changed for debugging
StartCommit = 0


#open file
#Results = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/BCofNodesByEdit-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'r')
Results = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/BCofNodesByEdit-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'r')
ResultsLines = Results.readlines()
Output = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/2.BCvaluesByRev-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')

resultsList=[]
for line in ResultsLines:
    resultsList.append(line.split('\t'))
    #    Results.write( str(revNum)+'\t'+nodename +'\t' +str(rank_bc.index(nodename)+1)+'/'+str(len(rank_bc)) +'\t' +str(bc[nodename])+'\n')

previousRev=0
numnodes=0
add=0
high=0
low = 0
graphsize=0
XY=[]

#first line in the file
Output.write('revnum\t#nodes\tadd\thigh\tlow\tgraphsize\n')

#get the pairs of RevNum and DevDays
dbcursor.execute("select commitNum, devdays from rev_commit order by commitNum")
Rev2Days = dbcursor.fetchall()

for item in resultsList:
    if item[0] == previousRev:
        if item[3].strip('\n')=='0.0':   #if zero, floor that value to the bottom end of the range
                                    #This is to avoid the fact that all zeros also have an apparent order. When there are too many zeros
                                    #then some of them might have a high rank order.
            add = add + int(item[2].split('/')[1])
            high = min (high, int(item[2].split('/')[1]))
            low = max (low, int(item[2].split('/')[1]))
        else:
            add = add + int(item[2].split('/')[0])
            high = min (high, int(item[2].split('/')[0]))
            low = max (low, int(item[2].split('/')[0]))
        numnodes = numnodes +1
    else:
        #write previous group results
        if previousRev != 0:
            Output.write(str(previousRev)+'\t'+str(numnodes)+'\t'+str(add)+'\t'+str(high)+'\t'+str(low)+'\t'+str(graphsize)+'\n')
            AvgInRange = 1-(float(add)/float(numnodes))/graphsize
            #what day is that rev?
            for d in Rev2Days:
                if int(d[0])==int(previousRev):
                    previousDay= d[1]
                    break
            XY.append([int(previousDay), AvgInRange])

        #prepare new set
        previousRev = item[0]
        numnodes=1
        add = int(item[2].split('/')[0])
        high = int(item[2].split('/')[0])
        low = int(item[2].split('/')[0])
        graphsize = int(item[2].split('/')[1])

Results.close()
Output.close()

print"Values calculated. Generating heatmap with "+str(len(XY))+" elements"
dotsize=40
saturation = 128 #from 0 to 255. 0 will saturate with just one dot, 255 will not register anything

imgheight = 1024
imgwidth = 1280
target = '../3. Results/33-H5-BetweenessCentrality/'+project+'/2.MSwide-DevDays-Heatmap-'+str(StartCommit)+'-'+str(TopCommit)+'-dot'+str(dotsize)+'-sat'+str(saturation)

topdot = (imgheight+dotsize/2.0)/imgheight
XY.append([0,topdot]) #add one dot to the top left corner in order to allow the dots in vertical 1 to be shown entirely
XY.append([0,1])


# add milestones
#milestones = [473, 1160,1976, 3826, 4947, 8732,  15309, 16536] #chandler
#milestones = [183, 309, 545, 1085, 2436, 3800, 7236] #trac
#milestones = [] #twisted (can't find milestones)
milestones = [30774, 41458, 70507, 87383, 105188] #zope

#what day is that rev?
for m in milestones:
    for d in Rev2Days:
        if int(d[0])>=int(m):
            XY.append([d[1],topdot])
            break


hm = heatmap.Heatmap()
hm.heatmap(XY, target, dotsize, size=(imgwidth,imgheight), scheme='fire', dotsaturation=saturation)

current = time.localtime()
print str(time.mktime(current)-time.mktime(start))

```

34-H5-PlotResults & HeatMap (AVG)_revs_as_time_milestones_wide.py

```
'''
Create the heatmap relating changes and the normalized rank order of betweeness centrality value of the files changed in each revision

'''
import MySQLdb
import time
import heatmap
import random #this is only required for the random verification, can be omitted

# Project
project='project'


##########################################
# initialize vars
start = time.localtime()

# Step 0 - Setup
# connect to db
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

#get how many revisions we have to process
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])
#TopCommit = 16627 #last commit for chandler
#TopCommit = 5000 # overwritten changed for debugging
StartCommit = 0


#open file
#Results = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/BCofNodesByEdit-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'r')
Results = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/BCofNodesByEdit-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'r')
ResultsLines = Results.readlines()
Output = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/2.BCvaluesByRev-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')
XY_Output = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/XY_Output.txt', 'w')

resultsList=[]
for line in ResultsLines:
    resultsList.append(line.split('\t'))
    #    Results.write( str(revNum)+'\t'+nodename +'\t' +str(rank_bc.index(nodename)+1)+'/'+str(len(rank_bc)) +'\t' +str(bc[nodename])+'\n')

previousRev=0
numnodes=0
add=0
high=0
low = 0
graphsize=0
XY=[]

#first line in the file
Output.write('revnum\t#nodes\tadd\thigh\tlow\tgraphsize\n')


for item in resultsList:
    if item[0] == previousRev:
        if item[3].strip('\n')=='0.0':   #if zero, floor that value to the bottom end of the range
                            #This is to avoid the fact that all zeros also have an apparent order. When there are too many zeros
                            #then some of them might have a high rank order.
            add = add + int(item[2].split('/')[1])
            high = min (high, int(item[2].split('/')[1]))
            low = max (low, int(item[2].split('/')[1]))
        else:
            add = add + int(item[2].split('/')[0])
            high = min (high, int(item[2].split('/')[0]))
            low = max (low, int(item[2].split('/')[0]))
        numnodes = numnodes +1
    else:
        #write previous group results
        if previousRev != 0:
            Output.write(str(previousRev)+'\t'+str(numnodes)+'\t'+str(add)+'\t'+str(high)+'\t'+str(low)+'\t'+str(graphsize)+'\n')
            AvgInRange = 1-(float(add)/float(numnodes))/graphsize
            #XY.append([int(previousRev), AvgInRange]) # this line to do the project analysis
            XY.append([int(previousRev), random.random()]) #This line to get the random test
        #prepare new set
        previousRev = item[0]
        numnodes=1
        add = int(item[2].split('/')[0])
        high = int(item[2].split('/')[0])
        low = int(item[2].split('/')[0])
        graphsize = int(item[2].split('/')[1])

Results.close()
Output.close()

print"Values calculated. Generating heatmap with "+str(len(XY))+" elements"
dotsize=40
saturation = 128 * len(XY)/9207 #from 0 to 255. 0 will saturate with just one dot, 255 will not register anything
                                #9207 is a factor borrowed from twisted. it is the len(XY) in that project
saturation = 128
imgheight = 1024
imgwidth = 1280
target = '../3. Results/33-H5-BetweenessCentrality/'+project+'/2.Random_MSwide-Heatmap-'+str(StartCommit)+'-'+str(TopCommit)+'-dot'+str(dotsize)+'-sat'+str(saturation)

#Write out a XY list
for item in XY:
    XY_Output.write(str(item[0])+'\t'+str(item[1])+'\n')
XY_Output.close()

#
topdot = (imgheight+dotsize/2.0)/imgheight
XY.append([0,topdot]) #add one dot to the top left corner in order to allow the dots in vertical 1 to be shown entirely

# add milestones
#milestones = [473, 1160,1976, 3826, 4947, 8732,  15309, 16536] #chandler
milestones = [183, 309, 545, 1085, 2436, 3800, 7236] #trac
#milestones = [] #twisted (can't find milestones)
#milestones = [30774, 41458, 70507, 87383, 105188] #zope

for i in milestones:
    XY.append([i,topdot])

#pack all the zope entries into a gapless count
if project=='zope':
    a=0
    XYzope=[]
    for i in XY:
       a +=1
       XYzope.append([a,i[1]])


hm = heatmap.Heatmap()
if project =='zope':
    hm.heatmap(XYzope, target, dotsize, size=(imgwidth,imgheight), scheme='fire', dotsaturation=saturation)
else:
    hm.heatmap(XY, target, dotsize, size=(imgwidth,imgheight), scheme='fire', dotsaturation=saturation)

current = time.localtime()
print str(time.mktime(current)-time.mktime(start))
```

35-H5-RelativeChanges_OverPoint8.py

```python

import MySQLdb
import time
import heatmap

# Project
project='zope'


##########################################
# initialize vars
start = time.localtime()

# Step 0 - Setup
# connect to db
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

#get how many revisions we have to process
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])
#TopCommit = 16627 #last commit for chandler
#TopCommit = 5000 # overwritten changed for debugging
StartCommit = 0


#open file
#Results = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/BCofNodesByEdit-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'r')
Results = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/BCofNodesByEdit-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'r')
ResultsLines = Results.readlines()
Output = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/2.BCvaluesByRev-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')
Output2 = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/3.Point8_Ratio_win60-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'w')

resultsList=[]
for line in ResultsLines:
    resultsList.append(line.split('\t'))
    #    Results.write( str(revNum)+'\t'+nodename +'\t' +str(rank_bc.index(nodename)+1)+'/'+str(len(rank_bc)) +'\t' +str(bc[nodename])+'\n')

previousRev=0
numnodes=0
add=0
high=0
low = 0
graphsize=0
XY=[]

#first line in the file
Output.write('revnum\t#nodes\tadd\thigh\tlow\tgraphsize\n')

#get the pairs of RevNum and DevDays
dbcursor.execute("select commitNum, devdays from rev_commit order by commitNum")
Rev2Days = dbcursor.fetchall()

for item in resultsList:
    if item[0] == previousRev:
        if item[3].strip('\n')=='0.0':   #if zero, floor that value to the bottom end of the range
                            #This is to avoid the fact that all zeros also have an apparent order. When there are too many zeros
                            #then some of them might have a high rank order.
            add = add + int(item[2].split('/')[1])
            high = min (high, int(item[2].split('/')[1]))
            low = max (low, int(item[2].split('/')[1]))
        else:
            add = add + int(item[2].split('/')[0])
            high = min (high, int(item[2].split('/')[0]))
            low = max (low, int(item[2].split('/')[0]))
        numnodes = numnodes +1
    else:
        #write previous group results
        if previousRev != 0:
            Output.write(str(previousRev)+'\t'+str(numnodes)+'\t'+str(add)+'\t'+str(high)+'\t'
+str(low)+'\t'+str(graphsize)+'\n')
            AvgInRange = 1-(float(add)/float(numnodes))/graphsize
            #what day is that rev?
            for d in Rev2Days:
                if int(d[0])==int(previousRev):
                    previousDay= d[1]
                    break
            XY.append([int(previousDay), AvgInRange])

        #prepare new set
        previousRev = item[0]
        numnodes=1
        add = int(item[2].split('/')[0])
        high = int(item[2].split('/')[0])
        low = int(item[2].split('/')[0])
        graphsize = int(item[2].split('/')[1])

Results.close()
Output.close()

print"Values calculated. Generating heatmap with "+str(len(XY))+" elements"
dotsize=40
saturation = 128 #from 0 to 255. 0 will saturate with just one dot, 255 will not register anything
imgsize = 1024
target = '../3. Results/33-H5-BetweenessCentrality/'+project+'/2.HeatmapPack-'+str(StartCommit)+'-'+str(TopCommit)+'-dot'+str(dotsize)+'-sat'+str(saturation)

#pack all the zope entries into a gapless count
a=0
XYzope=[]
for i in XY:
   a +=1
   XYzope.append([a,i[1]])


XYzope.append([0,(imgsize+dotsize/2.0)/imgsize]) #add one dot to the top left corner in order to allow the dots in vertical 1 to be shown entirely
XYzope.append([0,1])

hm = heatmap.Heatmap()
hm.heatmap(XYzope, target, dotsize, size=(imgsize,imgsize), scheme='fire', dotsaturation=saturation)


#how many are over .8 betweeness centrality
bins=[]
for i in range(0,80):
    bins.append([0,0])

for item in XY:
    bins[item[0]/60][0] +=1
    if item[1]>0.8:
        bins[item[0]/60][1]=int(bins[item[0]/60][1])+1

for t in bins:
        Output2.write(str(t[0])+'\t'+str(t[1])+'\n')

Output2.close()

current = time.localtime()
print str(time.mktime(current)-time.mktime(start))

```

38-H6-NewOrOldCore-Point8_time.py

```
# Project
project='project'

##########################################
import time
import MySQLdb
import heatmap

# initialize vars
start = time.localtime()

# Step 0 - Setup
# connect to db
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

#get how many revisions we have to process
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])
#TopCommit = 16627 #last commit for chandler
#TopCommit = 5000 # overwritten changed for debugging
StartCommit = 0

#get the pairs of RevNum and DevDays in a dictionary
Rev2Days = {}
dbcursor.execute("select commitNum, devdays from rev_commit order by commitNum")
Rev2DaysDB = dbcursor.fetchall()
for d in Rev2DaysDB:
    Rev2Days[d[0]] = d[1]

#open file
#Results = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/BCofNodesByEdit-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'r')
Results = open ('../3. Results/33-H5-BetweenessCentrality/'+project+'/BCofNodesByEdit-'+str(StartCommit)+'-'+str(TopCommit)+'.txt', 'r')
ResultsLines = Results.readlines()
Results.close()
Output = open ('../3. Results/38-H6-NewOrOldCore/'+project+'/38-H6-NewOrOldCore-AbovePoint8_time'+project+'.txt', 'w')


resultsList=[]
for line in ResultsLines:
    resultsList.append(line.split('\t'))

#create date of birth dictionary
dob = {}
for item in resultsList:
    if item[1] not in dob:
        dob[item[1]]=Rev2Days[int(item[0])]

#create the time to age pairs
Output.write('time\tage\tnormalized\n')
XY=[]
for item in resultsList:
    if float(eval(item[2]))<1/3.0 and Rev2Days[int(item[0])] > 0:
        Output.write(str(Rev2Days[int(item[0])])+'\t'+str(Rev2Days[int(item[0])]-dob[item[1]])+'\t'+str((Rev2Days[int(item[0])]-dob[item[1]])/Rev2Days[int(item[0])])+'\n')
        #Output.write(str(Rev2Days[int(item[0])])+'\t'+str(Rev2Days[int(item[0])]-dob[item[1]])+'\t'+'\n')
        XY.append([int(item[0]), int(item[0])-int(dob[item[1]])])

Output.close()
current = time.localtime()
print str(time.mktime(current)-time.mktime(start))

```

Extractor.py

```
'''
******************************************************************************************************
Extract calls from a given file

Global module flag outputs 'GlobalModule' as the library being called to any library that is part of the
    python global module list

Only python source files supported

From Python documentation (http://www.python.org/doc/2.5.2/ref/import.html)
**************************************
import_stmt  ::=  "import" module ["as" name] ( "," module ["as" name] )*
                | "from" relative_module "import" identifier ["as" name]
                  ( "," identifier ["as" name] )*
                | "from" relative_module "import" "(" identifier ["as" name]
                  ( "," identifier ["as" name] )* [","] ")"
                | "from" module "import" "*"
**************************************

Returns a list with the lines that match the call patterns
'''

def ExtractCallsIgnoreComments(FilePath, GlobalMod_ON=0):
    import re
    output=[]
    INcomment = False

    #test if file is in a programming language that is supported here
    if not FilePath.endswith('.py'):
            return output

    #load global module list
    if GlobalMod_ON==1:
        listaGlobal=[]
        f=open('PythonGlobalModuleIndex.txt', 'r')
        fileList = f.readlines()
        for fileLine in fileList:
            fileLine = fileLine.replace ( "\n", "" )
            listaGlobal.append(fileLine)
        f.close()

    fileHandle = open (FilePath, 'rU')  # the code rU reads lines that have \r (carriage return) as also being \n (newline)
                                        # so that the file.readlines() works correctly
    fileList = fileHandle.readlines()
    for lineindex in range(0,len(fileList)):
        fileLine = fileList[lineindex].replace ( "\n", "" )
        fileLine = fileLine.strip() #remove whitespace on the line like spaces and tabs at beginning of line

        if fileLine.find("'''")>-1 or fileLine.find('"""')>-1: #if this line starts or ends a comment section
            numOfBracks = max(len(fileLine.split("'''")) , len(fileLine.split('"""')))
            for i in range (0,numOfBracks-1):
                INcomment = not INcomment#caveat: ignoring import statements in between of comments
            continue #skip to the next iteration of the loop, otherwise the commented line would be processed

        if not INcomment:
            fromData = re.match('(from .*)',fileLine)
            if (fromData is not None):
                temp = fileLine.replace("  "," ").split(" ") #split around the space
                output.append([temp[1],lineindex])
                #print [temp[1],lineindex]
                temp=''

            fromData = re.match('(import .*)',fileLine.lower())
            if (fromData is not None):
                fileLine=fileLine.replace("  "," ")
                fileLine=fileLine.split(",") # split around the commas
                for i in range(0, len(fileLine)):
                    if GlobalMod_ON==1:
                        try:
                            if listaGlobal.index(fileLine[i]):  #found global
                                fileLine[i]='GlobalModule'      #changed name to GlobalModule
                        except:                                 #did not find global
                            pass
                    fileLine[i] = fileLine[i].lstrip().split(" ")
                    if fileLine[i][0].lower() == 'import':
                        output.append([fileLine[i][1],lineindex])
                        #print [fileLine[i][1],lineindex]
                    else:
                        output.append([fileLine[i][0],lineindex])
                        #print [fileLine[i][0],lineindex]
        fileLine="" #reset the variable for the next loop, otherwise the for cycle may want to
a len(fileLine) too big

    fileHandle.close()
    return output


```

Path.py

```

'''
Distance - Calculates the path length between two files, given a folder path
****
The algorithm consists of pruning the path, starting from the root, up to where
the paths have a difference.
Then we can simply add the number of remaining elements to get the distance between
the two items.
'''
def distance(file1, file2):
    file1 = file1.split('/')
    file2 = file2.split('/')

    while file1[0] == file2[0]:
        file1.pop(0)
        file2.pop(0)

        if len(file1)==0 or len(file2)==0:
            break

    #print 'distance = '+str(len(file1)+len(file2))
    return (len(file1)+len(file2))


'''
Reduces a path+filename to a simple name
****
Keeps a history of reductions made. Verifies if reduction has been made in the past
or if the reduced name has been used by another object
'''
def reduce(path,project,useDupeLogs=1,revnum=0):
    import os

    dupes = 0
    reduceLogList = [] # the reductions we have already done
    dupeLogListPaths = []
    reduceLogLines = []
    resultsUsed = []

    #reduce the path
    tempresult = path.replace('.py','')
    tempresult = tempresult.split('/')[-1] #returns the last element in the list after split
on the "/"

    if useDupeLogs==1:
        #create files for the first time in a project
        if not os.path.isfile ('reduceLog_'+project+'.txt'):
            reduceLog = open('reduceLog_'+project+'.txt','w')
            reduceLog.close()
        if not os.path.isfile ('reduceLogDupes_'+project+'.txt'):
            reduceLogDupes = open('reduceLogDupes_'+project+'.txt','w')
            reduceLogDupes.close()

        #open those log files
        reduceLog = open('reduceLog_'+project+'.txt','r+a')
        reduceLogDupes = open('reduceLogDupes_'+project+'.txt','r+a')

        # load the reduceLog into the source and target lists
        reduceLogLines = reduceLog.readlines()
        if len(reduceLogLines)>0: #the first time around it's empty...
            for i in range (0, len(reduceLogLines)):
                #the lines is formatted: revnum \t path \t reduce\n
                reduceLogList.append([reduceLogLines[i].split('\t')[1],reduceLogLines[i].split
('\t')[2].replace('\n','')])


        # test if we already had that path and get it's value (it might be different than
        # tempresult as it may belong to a dupe calculated in a previous pass. if so, then
take
        # that result
        for pair in reduceLogList:
            if pair[0]==path:
                result = tempresult = pair[1]
                break

        if [path, tempresult] not in reduceLogList:
            #got a new pair
            ##print 'got new pair'
            #check if just the result value isn't being used by other path
            found=0
            for pair in reduceLogList:
                if pair[1]==tempresult:
                    found=1
                    break

            # if just the result wasn't found, then it is an entirely new pair
            if found ==0:
                result = tempresult
                if int(revnum)>0:
                    reduceLog.write(str(revnum)+'\t')
                reduceLog.write(path+'\t'+result+'\n')

            # if the result was already being used in another pair
            if found ==1:
                dupes = dupes +1
                #register it in the dupesLog if it doesn't exist there already
                #read the file in
                dupeLogList = reduceLogDupes.readlines()
                #get the dupe paths
                for i in range (0, len(dupeLogList)):
                    dupeLogListPaths.append(dupeLogList[i].split('\t')[2].replace('\n',''))

                #if the current dupe path is not on that list, then we add it
                if path not in dupeLogListPaths:
                    if int(revnum)>0:
                        reduceLogDupes.write(str(revnum)+'\t')
                    reduceLogDupes.write(tempresult+'\t'+pair[0]+'\t'+path+'\n') #pair[0] was the first path to claim the result

                #generate a new result for this dupe path by adding '-JC'
                for line in reduceLogLines:
                    resultsUsed.append(line.split('\t')[2].replace('\n',''))
                result = tempresult+'-JC'
                while result in resultsUsed:
                    result = result+'-JC'

                reduceLog.write(str(revnum)+'\t')

                reduceLog.write(path+'\t'+result+'\n')

        else:
            #already had this one in the list, nothing changes
            result = tempresult
            ##print 'already had: '+path+','+result

        #print dupes

        reduceLog.close()
        reduceLogDupes.close()

    else:
        result = tempresult

    return result

'''
Reduces a path+filename to a path.name
****
Keeps a history of reductions made. Verifies if reduction has been made in the past
or if the reduced name has been used by another object
'''
def reducedot(path,project,useDupeLogs=1,revnum=0):
    import os

    dupes = 0
    reduceLogList = [] # the reductions we have already done
    dupeLogListPaths = []
    reduceLogLines = []
    resultsUsed = []

    #reduce the path. Example: '../svncheckout/chandler/trunk/chandler/application/repository/simple_test.py'
    tempresult = path.replace('.py','')    #example: ../svncheckout/chandler/trunk/chandler/application/repository/simple_test
    tempresult = tempresult.replace('../svncheckout/'+project+'/trunk/','') #used to be>> tempresult = tempresult.replace('../svncheckout/chandler/trunk/','')
    tempresult = tempresult.replace('c:/svncheckout/'+project+'/trunk/','')    #used to be>> tempresult = tempresult.replace('c:/svncheckout/chandler/trunk/','')
    tempresult = tempresult.replace('c:/svncheckout/zope/Zope/trunk/','') #specific for the
```

```
way Zope is set up
    tempresult = tempresult.replace('/home/joao/Documents/svncheckout/'+project+'/trunk/','')
    tempresult = tempresult.replace('/','.').lower()

    if useDupeLogs==1:
        #create files for the first time in a project
        if not os.path.isfile ('reduceLog_'+project+'.txt'):
            reduceLog = open('reduceLog_'+project+'.txt','w')
            reduceLog.close()
        if not os.path.isfile ('reduceLogDupes_'+project+'.txt'):
            reduceLogDupes = open('reduceLogDupes_'+project+'.txt','w')
            reduceLogDupes.close()

        #open those log files
        reduceLog = open('reduceLog_'+project+'.txt','r+a')
        reduceLogDupes = open('reduceLogDupes_'+project+'.txt','r+a')

        # load the reduceLog into the source and target lists
        reduceLogLines = reduceLog.readlines()
        if len(reduceLogLines)>0: #the first time around it's empty...
            for i in range (0, len(reduceLogLines)):
                #the lines is formatted: revnum \t path \t reduce\n
                reduceLogList.append([reduceLogLines[i].split('\t')[1],reduceLogLines[i].split('\t')[2].replace('\n','')])


        # test if we already had that path and get it's value (it might be different than
        # tempresult as it may belong to a dupe calculated in a previous pass. if so, then take
        # that result
        for pair in reduceLogList:
            if pair[0]==path:
                result = tempresult = pair[1]
                break

        if [path, tempresult] not in reduceLogList:
            #got a new pair
            ##print 'got new pair'
            #check if just the result value isn't being used by other path
            found=0
            for pair in reduceLogList:
                if pair[1]==tempresult:
                    found=1
                    break

            # if just the result wasn't found, then it is an entirely new pair
            if found ==0:
                result = tempresult
                if int(revnum)>0:
                    reduceLog.write(str(revnum)+'\t')
                reduceLog.write(path+'\t'+result+'\n')

            # if the result was already being used in another pair
            if found ==1:
                dupes = dupes +1
                #register it in the dupesLog if it doesn't exist there already
                #read the file in
                dupeLogList = reduceLogDupes.readlines()
                #get the dupe paths
                for i in range (0, len(dupeLogList)):
                    dupeLogListPaths.append(dupeLogList[i].split('\t')[2].replace('\n',''))

                #if the current dupe path is not on that list, then we add it
                if path not in dupeLogListPaths:
                    if int(revnum)>0:
                        reduceLogDupes.write(str(revnum)+'\t')
                    reduceLogDupes.write(tempresult+'\t'+pair[0]+'\t'+path+'\n') #pair[0] was the first path to claim the result

                #generate a new result for this dupe path by adding '-JC'
                for line in reduceLogLines:
                    resultsUsed.append(line.split('\t')[2].replace('\n',''))
                result = tempresult+'-JC'
                while result in resultsUsed:
                    result = result+'-JC'

                reduceLog.write(str(revnum)+'\t')

                reduceLog.write(path+'\t'+result+'\n')

        else:
            #already had this one in the list, nothing changes
            result = tempresult
            ##print 'already had: '+path+','+result

        #print dupes

        reduceLog.close()
        reduceLogDupes.close()

    else:
        result = tempresult

    return result
```

22-NumberOfFilesEditedPerRevisionHistogram.py

```
'''
For each revision counts how many files were changed and builds a histogram.

'''

import MySQLdb

# Project
project='twisted'
MaxCount=50
StartRev=1

#initialize variables
index=0
progress=0
editCounts=[0]
MoreThanMaxCount =0

print 'Starting work on: '+project

#Create result vector
for i in range (0,MaxCount):
    editCounts.append(0)

# Open connection to database
db = MySQLdb.connect(host="localhost", user="username", passwd="password", db=project)
dbcursor = db.cursor()

#top commit
dbcursor.execute("select commitNum from rev_commit order by commitNum desc limit 1")
TopCommit = dbcursor.fetchall()
TopCommit = int(TopCommit[0][0])

for rev in range(StartRev,TopCommit+1):
    NumberOfFilesEdited=0
    # get all the filenames and insert them in a dictionary.
    # dictionary includes the index number that will be used as the table index for that file
    dbcursor.execute("select file from rev_files where commitNum="+str(rev))
    FilesEdited = dbcursor.fetchall()

    for item in FilesEdited:

        if item[0].endswith('.py'):
            NumberOfFilesEdited +=1

    if NumberOfFilesEdited>MaxCount:
            MoreThanMaxCount += 1
            #print '# of MoreThanMaxCount:'+str(MoreThanMaxCount)+' Rev:'+str(rev)+'
Value:'+str(NumberOfFilesEdited[0][0])
    else:
        editCounts[NumberOfFilesEdited] += 1

    if rev %1000== 0:
        print rev



f=open('NumberOfFilesEditedPerRevisionHistogram_'+project+'_output.txt', 'w')
for i in range (1,MaxCount):
    f.write(str(editCounts[i])+'\n')
f.write('\nMoreThanMaxCount '+str(MoreThanMaxCount))
f.close()

print 'MoreThanMaxCount: '+str(MoreThanMaxCount)
print 'Done'
```

Yep. Done. The End. Thank you.