COMMUNICATION PATTERNS AND STRATEGIES IN SOFTWARE DEVELOPMENT COMMUNITIES OF PRACTICE

By

Shreya Kumar

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2016

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

Dissertation Advisor: *Charles Wallace*

Committee Member: *Linda Ott*

Committee Member: *Mary Beth Rosson*

Committee Member: *Janet Burge*

Committee Member: *Lauren Bowen*

Department Chair: *Min Song*

To Mom, Dad and Shraddha,
for loving and encouraging me from across the globe,
even though it meant more time apart.

To my loving husband, Randy,
for constantly supporting me and lifting my spirits,
for giving me a reason to persevere everyday.

Contents

List of Figures

List of Tables

Preface

Chapter 2 is a modified version of the paper *Communication Strategies for mentoring in software development projects* by Kumar and Wallace, presented at the 44th Annual Frontiers in Education (FIE) IEEE Conference, 2014. Kumar and Wallace co-wrote the paper and collaboratively performed the data analysis after Kumar collected the data.

Chapter 3 is a modified version of the paper *Among the agilists: Participant observation in a rapidly evolving workplace(in press)* by Kumar and Wallace to be presented at the 9th International Workshop on Cooperative and Human Aspects of Software Engineering(CHASE), 2016. Kumar and Wallace collaboratively wrote the paper based on Kumar's experience and data collected as a participant observer.

Chapter 6 is a combination of two published papers. Sections 6.1 is a heavily modified version of part of the paper *Agile Communicators: Cognitive Apprenticeship to Prepare Students for Communication-Intensive Software Development* by Kumar, Ureel, and Wallace, presented at the Agile 2015 conference. This paper was collectively written by Kumar, Ureel and Wallace. Kumar and Ureel contributed their experiences of experimenting with their modified computing courses that were designed with Wallace, who brought the whole work under the Agile umbrella. Section 6.2 is a modified version of the paper *Instruction in Software Project Communication through Guided Inquiry(in press)* by Kumar and Wallace, to be presented at the 38th International Conference on Software Engineering, 2016. Kumar and Wallace wrote the paper together based on the specifics of their communication based Team Software Project course.

Acknowledgments

This work would not have been possible without the contributions of a vast support network.

My advisor, Dr. Charles Wallace For his constant guidance, encouragement and patience. For creating an atmosphere where I felt excited to wake up and work every day. For making this a rewarding journey.

My committee members For agreeing to lend me their time and for their expertise in improving our research, encouragement and enthusiasm. For inspiring me with their groundbreaking work that we were able to leverage for our research.

TAI For allowing me to study their software development practices, making me feel welcome and enthusiastically supporting my research.

My husband, Randy For his unwavering love and support. For his constant encouragement and for never letting me feel alone in this process.

My family and friends Who kept me sane and motivated, reminding me that there is light at the end of the tunnel.

Abstract

Some of the greatest challenges in the relatively new field of software development lie in the decidedly old technology of communication between humans. Software projects require sophisticated and varied communication skills because software developers work in a world of incomplete, imperfect information where teams evolve rapidly in response to evolving requirements and changing collaborators. While prescriptive models for software process such as Agile suggest ways of doing, in reality these codified practices must adapt to the complexities of a real workplace. Patterns, rather than rules of behavior within software process are more suitable to the varied and mutable nature of software development. Software development communities are also learning communities, attempting to sustain themselves through internal ambiguity and external changes.

We study different types of software development communities to fulfill our goal of understanding how these communities implement and evolve different communication strategies to sustain themselves through change. We observe student software development projects, open source software development, and a professional, rigorously Agile software development community. We employ Wenger's concept of Community of Practice to frame our understanding, especially focusing on the notions of identity, participation, reification, negotiation of meaning and trajectory of the participants of the software development communities. From these different sources, we identify the emergent themes of mentoring and knowledge management as critical for sustainable communities.

Through our long running, immersive, participant observer, ethnographic study of the Agile software development community, we contribute both a quantitative and qualitative analysis of the their communication practices and depict the evolving nature of their onboarding and mentoring strategies. We share our experience of implementing such an immersive industry ethnographic study. We employ a pattern language approach to capturing, analyzing and representing our results, thereby contributing and relating to the larger bodies of work in Scrum and Organizational Patterns.

This work also informs our concurrent efforts to enhance our undergraduate computer science and software engineering curriculum, exposing students to the communication challenges of real software development and help them to develop skills to meet these challenges through practice in inquiry, critique and reflection.

Chapter 1

Introduction

In this chapter, we illustrate the complexities of software development communication through real world examples and discuss the background and framework, especially Communities of Practice, for our research. We present our research questions and we briefly describe our research subjects, tools and methodology. We describe our initial research efforts studying student software development projects and open source software development in chapter 2.

In chapter 3, we describe our experience of observing a dynamic, co-located software development community over a long term using immersive ethnography. We describe the methodology employed for them and the challenges faced in taking on the endeavor of long term participant observation. In chapters 4 and 5, we share our observations from the study and the pattern-form results from the ethnographic study.

In chapter 6, we share our efforts of integrating communication intensive coursework into the computing curriculum and plans to expand it. We summarize our research story, our primary takeaways and discuss future directions in chapter 7.

1.1 Communication in software development: Two vignettes

Communication in any software development community can be complex, nuanced and varied. Making a communication choice can have unintended consequences or can be a

strategy for successful interchange of ideas and unambiguous decision-making. We offer two examples of communication in software development, one from academia and one from industry.

Vignette 1:

Setting: As part of their senior capstone project, a team of three software engineering students are working on a project for the US Navy. Their client and technical expert is Hank Taylor, a professor in the Mechanical Engineering department. An earlier team had met with the client several times and tried but did not succeed in producing the code needed. The current three person software development team has taken over from them and is attempting to finish the project with Hank. Several weeks into the project, the current team finds itself facing similar issues to the first team - they are behind schedule in presenting a requested analysis of the existing legacy code.

In this vignette, Denise, one of the software engineering students shares a rough hand drawn data dependency and control flow chart (see Figure 1.1 on page 3) of the legacy code with Hank. They use the hand drawn chart to point at areas of the code represented by boxes, to be able to clearly articulate their questions and responses, using the chart as context for their questions. Together, they read the chart and mark it up as they go along.

```
Hank (client):  Is this your chart?

Denise (student CS team member):  Yes

Hank:  It looks exactly like his (another chart from a
company that is a navy contractor, to keep track of the
code)

Denise:  No, his chart is much nicer.

Hank:  So have you folks started 'divvying' it up?

Denise:  This is where we need some help.  So this is what
happens in the code (pointing at Denise's chart) [Denise
explains on her chart that she has color coded based on
which blocks are her responsibility and how the chart
describes the blocks]
```

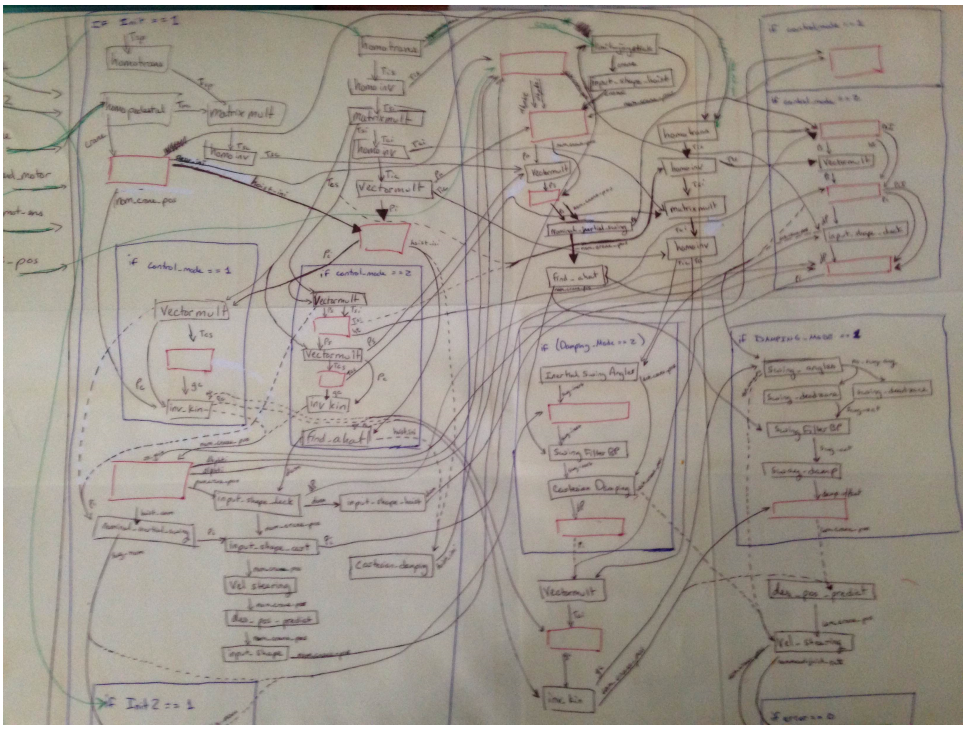

Figure 1.1: Denise's hand-drawn chart

Hank: Can you show me some example within the code? This is great. Don't throw this out. Is this hand-drawn?

Denise: Yes, I love the colors.

Vignette 2:

Setting: Audacity is a popular, open source, cross platform, recording and audio editing tool. The Audacity project has been running for many years and has been exercising software development communication using email as the primary means of communication and decision making. The project is profoundly distributed in nature, with members from different countries. They have a core team of four to six people and receive code contributions from an assortment of programmers. Anyone in the developer list can start a discussion. Some of the participants on the forum may be regular contributors but not on the Core Team. The participation ranges from extremely regular to sporadic for some of the developers. The replies on the forum can be complex in style and are often inline with varying degrees of quoting.

In the following email exchange, Campbell is an occasional contributor and Benjamin,

who has been regularly contributing for two years, has recently been inducted into the Core Team. Cambell brings up the question of code style and Benjamin enthusiastically agrees with him. Leland and Vaughn are long time contributors in the Core Team where Vaughn could be considered the most active on email. Vaughn disagrees with the suggestion and eventually reveals that this type of discussion has occurred before and decided.

The project has just wrapped up a code freeze and the developers have some time to reflect on the bigger picture. This is a discussion post code freeze time when developers have time to reflect on the bigger picture. Benjamin supports another new developer Campbell, who brought up the idea of following standard code style in the programming. The idea is debated where some relative newcomers are in support of imposing coding standards and some of the more experienced developers in the community are against the idea and do not want to continue the discussion as this issue had been discussed and decided before.

Campbell Barton wrote:

Hi,

I was curious if Audacity has a preferred style guide.

Richard Ash wrote:

Hmm, google: audacity code style

first hit

http://wiki.audacityteam.org/wiki/CodingStandards

(and link in the second section of the Developer Guide).

Benjamin Drung wrote:

The coding standard says:

three spaces per indent

Why? Every other software project that I know uses either two, four, eight spaces per indent or tabs. Some code/text editors do not support three spaces per indent.

Leland wrote:

Here's a little more background:

http://audacity.238276.n2.nabble.com/How-about.html

And I'm sure it was discussed way before that. For those that don't know Dominic was one of the original authors of Audacity.

Benjamin Drung says:

Maybe it's time to discuss it again. Old developers retired and new joined. So the overall preference could have been changed. Who of the Audacity developers has strong opinions regarding coding styles and who does not care (as long it's consistent)?

Three spaces for indentation was a compromise between two and four. Who is for two spaces and who is for four?

After several emails, Vaughn wrote in response to Leland:

On 2/8/2013 2:48 PM, Leland wrote:

On Fri, Feb 8, 2013 at 2:01 AM, Vaughan Johnson

We've had many developers over the years who participate for a short (sometimes long, often infrequent) time, then move on, or become lurkers. (Ahem, like the guy who started this thread, and admitted he'd diverged from prior style when he contributed - much love!)

If this is a reference to me, then I was just doing what I thought you wanted...to keep quiet.

(Vaughn:) Thanks, Leland. I appreciate that, too.

Your original response was quite clear and nothing more needed to be said. Basically, the Audacity project is flexible, to some extent, when it comes to coding style.

Why have you kept the thread alive? It's really simple...don't respond.

(Vaughn:) I kept alive to engage the new posters, who apparently weren't getting what I said in my original response, and diverged into other topics. Sorry it's not more pleasant, but I get frustrated repeating myself and being argued against the same thing. I'm glad you found it clear what I was saying, but yes, I did feed the troll.

Thanks, Vaughan

These seemingly ordinary scenarios are complex under the surface. They are a product of choices, conscious or unconscious, on the part of the participants. In the capstone project vignette, the use of an artifact, in this case Denise's chart which started off as a tool for developing her own understanding of the legacy code, to facilitate a design and code specific technical discussion is an innovative way to disambiguate questions and explanations about

code. The artifact also expresses to the client the ability of the student to understand the code. It gives the student a chance to show her work and inspire confidence in her abilities. When the same hand drawn chart was used in a class activity in Team Software Project as an artifact of software development, many students reacted to the chart by calling it "messy and unprofessional" and "something (we) would never use with a client". However, in this instance, Hank, the client was very impressed with the chart and in turn with Denise, the student. He exclaimed that now Denise is the person who knows the most about the code. Upon learning of the context some more, the capstone project students understood its place as an artifact for personal consumption, therefore messy, but being used to demonstrate knowledge to the client.

In the Audacity vignette, when Benjamin supported the question of enforcing style guidelines, he was taking initiative and encouraging a discussion. However, as the discussion ran very long and eventually became unpleasant, it did not serve the originator in expressing his vision and attempt to foster a healthy discussion. Instead, some of the more experienced developers in the community were annoyed at the persistence of the discussion that they were trying to resolve quickly.

Also, at one point Benjamin poses a question about who supports which decision regarding coding style. "Three spaces for indentation was a compromise between two and four. Who is for two spaces and who is for four? " This is an awkward way to move the discussion forward before the more mature developers on the project have even given assent. The first issue with this type of statement is that it is a premature challenge to the status quo of no specific coding style and it calls for broad input which is awkward in a medium like email, where a specific question about who supports 2 spaces versus who supports 4 spaces for indentation, when the current standard is 3 spaces. This raises the questions: what would be a good alternative to the poll taking, and how does one determine that the poll taking is done? In this project, email poll taking is a common practice. However, questions do not often get more than a few responses, and those responses can be a factor in the decision, dependent on who supports it.

These are both examples of communication that have a significance beyond just the simple act of the communication, influenced by the context. Both are cases of communication which is strategically planned overall – planning a meeting, preparing a chart, starting a discussion about code style; however the participants have to improvise the specifics of the conversation, thinking tactically. The details of the communication require consideration, and there is no template or process that participants could follow to determine that. They can plan what the activity is going to be, and while engaged in the activity, they make smaller choices about their communication and can also have a great impact on the outcome of their communication.

1.2 Speech Act Theory: Speaking and Writing as Doing

Software engineers are accustomed to the idea of text having more power than the simple words used. Similarly, spoken and written words can have significance beyond the mere text. This brings us to a powerful idea that we do things by simply saying and writing. Software developers in an implicit way have to consider the impact of their words, which can lead to problems if not considered carefully. Following Austin's speech act theory [4] we can think of the communication acts as having a *locutionary* significance – with the utterance of the questions and answers in the student- client discussion, or the sending of the email in the open source community and the simple, immediate and direct meaning of the communication; as well as an *illocutionary* force – where the social significance of the utterance is considered, with the client asking questions or Benjamin making a suggestion as a social act of expressing or questioning for the benefit of the receiver; and the *perlocutionary* significance of the act where the actual effect of the utterance is considered, where Denise gets a chance to prove her knowledge or prove that she has done the work asked of her, or Benjamin is encouraging discussion with a simple suggestion about code style but also showing that he is thinking about the bigger picture as an active and core member of the team.

1.3 Communication in Software Development

There is a consensus among software professionals that some of the biggest challenges in this relatively new field lie in the decidedly old technology of communication - oral, written, and otherwise - between humans. Software development is complex, due not only to the functionality of the software itself, but also to the competing and often conflicting goals of different stakeholders. The wide range of application areas draws together stakeholders with different backgrounds. Moreover, software developers work in a world of incomplete, imperfect information, and they must be proactive in seeking input from other stakeholders [10]. Among software professionals, the quality of team communication is widely acknowledged to be a key factor in the success of failure of software projects.

In most real software settings, there is no comprehensive, rote communication workflow to follow. Developers must be able to think strategically and tactically about their communication, selecting the appropriate material, location, timing and manner to suit the context. This involves analysis of the context and of candidate solutions, along with a synthesis of communication elements. Agile development methods in particular stress the importance of flexible communication practices, deployed as the developers see fit (in contrast to the

static organization and communication practices of waterfall development).

The essential intangibility and malleability of software present unique communication challenges in its development, compared to other engineered products. The intangibility of software mandates sophisticated communication to convey requirements and changes, in ways that accommodate multiple stakeholders' mental models and goals. The malleability of software leads stakeholders to expect change at low cost with high impact. The software industry is characterized by constant personnel turnover, making software development even more dynamic. Consequently, the meaning of the software is something that needs to be negotiated continually. The members of a software development community must be empowered to convey and negotiate meaning through effective communication practices.

1.3.1 Communication and Software Process

In the early days of software engineering, the first decade after the term was coined at a seminal NATO meeting in 1968 [48] the notion of software process and the importance of communication in the process were emergent ideas. Code bases were smaller and even developer communities were smaller, often there were few precedents to follow in designing system behavior or interface. Disasters like the Therac-25 incident [46] occurred about two decades after the NATO conference. The Therac-25 causes can be traced back to a lack of thought or rigor in creating a reproducible process and lack of an overarching structure to the software process. This started to motivate the need for more rigor and process in software development. Software was starting to become more complex and entering newer domains and being made for different types of users.

In his seminal work on software development, Royce [52] proposes the waterfall model as a theoretical basis for a software process. It is not clear how viable even Royce considered the model, but it served as a hypothetical model in the lack of anything better. He was driven by the need for better documentation and more communication. Figure 1.2 on page 10 represents the stages of the software process and the documentation produced as a linear sequence of steps with limited backtracking. As other software development models were introduced (spiral, iterative, etc.), they were all based on adding structure to the communication and documentation practices, emphasizing the importance and impact of communication in the software development process. What is interesting and often ignored when discussing waterfall, is how many formal documents are produced at the end of each step. It is however, left unclear what one is supposed to do within each step beyond generating documents.

The 1980s and 90s witnessed a move away from a strictly linear structure of the waterfall

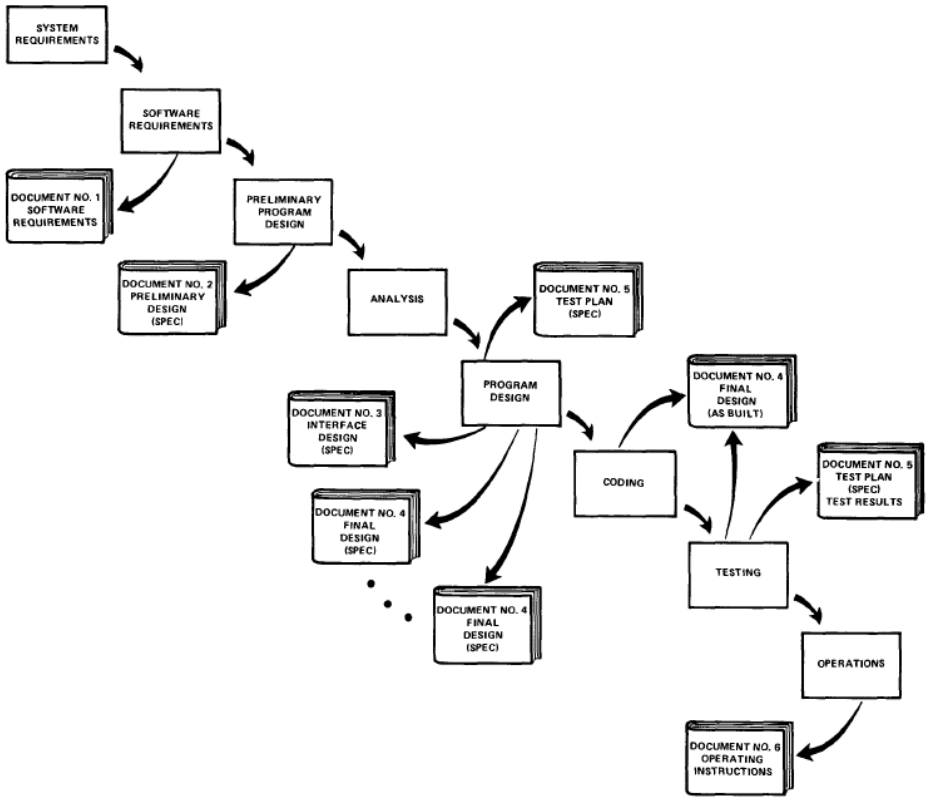

Figure 1.2: Waterfall model with documentation

model and away from strictly document-oriented processes. Carroll's work on minimalist design [16] motivated thinking about software interfaces to assist the user's abilities instead of placing the cognitive burden of necessarily going through lengthy documentation on the user. There was less emphasis on the preferred template style communication prescribed by the waterfall model. XP[5] put forth the somewhat radical notion that the documents laid out in waterfall are not necessary and may even hinder progress on the code.

As Agile software development has emerged, supported by the Agile manifesto [6], it also emphasizes the need for change in communication practices within the team, exemplified by XP, and between the development team and the product owner. Scrum incorporates very structured communication practices like the Daily Stand Up meeting, the Sprint Retrospective meeting and the Burndown chart. The purpose is to value "Individuals and Interactions" and "collaboration". There is greater emphasis and value placed on flexibility (agility). We observe a shift from documentation driven waterfall models to more interaction driven Agile methodology.

The structured communication practices are a starting point for practices, not a cookie cutter solution. This is a step further from the cookie cutter template model of software

communication, instead the drive was to describe a way of doing something and leave room for improvisation and choosing style. This is illustrated by the Scrum Patterns [20] and the Organizational patterns work [22] where libraries of patterns of Scrum and Agile organizational practices and guidelines are used to prescribe overall behavior with room for picking which patterns apply best and using them as guidelines to determine practices for one's specific team. For example from the Scrum Patterns library, the "Developer Ordered Work Plan" pattern which depends on the "Sprint", "Sprint Backlog" and "Product Backlog" patterns and would benefit from employing the "Spirit of the Game" pattern. Similarly from the Organizational patterns library, the "Self-Selecting Team" pattern is related to the "Developer decides process" pattern, which define specific practices for self-managing software teams. We discuss these pattern libraries and their inter-relationships later in section 1.6.4. The industry has witnessed a change in its ability to produce software by changing the processes and procedures observed during software development.

1.3.2 Communication in Software Engineering Education

Instruction in communication has long been part of the computer science and software engineering curricula, but typically there has been a divide between the formal, highly technical documentation taught within the computing disciplines and instruction in teamwork and communication (invariably referred to as "soft skills") that is typically provided through ancillary courses. Instruction by experts in writing and communication has an essential place in computing education, but it must be matched with similar instruction within the computing disciplinary courses. Software engineering instructors are in a unique position to ground the material in authentic practices and validate communication by attending to it.

The importance of communication in the software process is beginning to be acknowledged in the software engineering education community. The most recent version of the Software Engineering Body of Knowledge (SWEBOK) [9] has an expanded treatment of communication, compared to previous versions, with breakout sections on "reading, understanding and summarizing", "writing", "team and group communication", and "presentation skills" – though still little in the way of specific recommendations.

> "Some communication can be accomplished in writing. Software documentation is a common substitute for direct interaction. Email is another but, although it is useful, it is not always enough; also, if one sends too many messages, it becomes difficult to identify the important information. Increasingly, organizations are using enterprise collaboration tools to share information. In

addition, the use of electronic information stores, accessible to all team members, for organizational policies, standards, common engineering procedures, and project-specific information, can be most beneficial." Page 11-11 [9]

We see that the report hints at the need for being more strategic about communication skills, but does not provide much information on what those strategies should be. This is a good observation and a move in the right direction, but there are no overlying principles developed. The daily communication is something that students still have to figure out for themselves. They get little instruction or practice in daily tactical project communication. This section of SWEBOK raises the concern about the design of the communication strategies, but there is little in place to prepare students for it.

A recent positive development is a project funded through the NSF CPATH program to bring the expertise of writing instructors into the computer science and software engineering curricula [29]. This project has produced a repository of assignments that engage students in authentic communication activities. Some effort has been made to categorize the genres of communication that arise in software development settings [15].

While these activities acknowledge that communication is important, they motivate the case for careful communication design and a treatment of communication as problem solving, which requires choosing from different forms of communication based on context, need and style. The challenge is – How do we place students in an authentic decision making role before they get to their senior capstone project?

1.4 Communities of Practice

Our topic deals with learning in complex, changing, social settings. Vygotsky's *social constructivism* gives us a way to view learning as a social activity. Vygotsky's notion of social constructivism of knowledge [62] talks about how information is processed into knowledge affected by the learner's social and cultural environment and how one's community is important for "negotiating meaning". Social constructivism stands in contrast to some positivist approaches of understanding learning which are based on the assumption that meaning is absolute and static. Positivist approaches are parodied through the idea of the Nurnberg funnel where the transfer of knowledge is presumed to happen from the teacher to the student in the form of facts and theory which the student consumes through a figurative funnel in their head. We favor Vygotsky's social constructivism and its interpretation of learning, which assumes that meaning is dynamic and is better conveyed in the social setting where it is created.

In the computing field, we often think in frames that treat meaning as static; however, code is only a manifestation of the negotiated collective understanding of requirements, arrived at through interaction among multiple stakeholders with different roles and involvement in the software development community. A community member's role, influence and contribution in a software development community of practice may also be negotiated over time as members join and leave and work in different capacities over time.

Some of the main challenges that software developers or software engineers face in industry are related to their lack of experience or knowledge in the soft skills related to communication and teamwork, as described in the works of Begel [7] and Begel and Simon [8] on novice software engineers. In addition, Hall et al.'s [32] research about employee turnover in software engineering projects describes the negative impact on the motivation in and success of a project. Zanetti [57] talks about the influence of social factors in turnover, where employee turnover is a reality of the typical software project experience. The software industry is a dynamic set of communities characterized by high turnover. Therefore, it is important to study the essence of how these software development communities manage and sustain themselves.

When one thinks of software development in an abstract form, one thinks of developers as a homogeneous group of skilled and experienced programmers following predetermined processes to deliver a product, however in practice, the developer community is always in flux as new members are learning the ropes and experienced members are taking on larger tasks or moving on to different roles. In addition, the meaning of software is always changing based on changing requirements and resource limitations.

We interpret communication in software projects as a negotiation of meaning. Wenger [67] captures this negotiated quality of meaning in his concept of Community of Practice. Wenger describes Communities of Practice as "groups of people who share a concern or a passion for something they do and learn how to do it better as they interact regularly". He views meaning as essentially linked with identity within a practitioner community. Those who identify as community members contribute to the shared negotiation of meaning, and participation in the negotiation of meaning reinforces identity.

Wenger says "Practice is about meaning as an experience of daily life"(pg. 52)[67]. He asserts that negotiation of this meaning or lived experience happens through participation and reification. That the experience of meaning occurs when established patterns of daily behavior are produced anew, allowing the pattern to either be confirmed or disproved or extended or replaced. This becomes the significance or negotiated meaning associated with that everyday practice. Wenger argues that all forms of engagement are a negotiation of meaning - the process by which we experience and engage the world.

Wenger describes the concept of participation as social engagement in a community as a member, where this experience is both personal and social. It can manifest itself as thinking, being, doing, talking, feeling of social relations and emotions. Our participation in a community shapes our experience of the world and in turn also shapes that community. Participation can take many forms and may be formally acknowledged or occur informally or even subconsciously. One may not perceive their participation as contributing anything to the community, but their mere participation helps shape that community in some way.

Wenger's notion of reification refers to either concrete products or conceptual products as well as the process of producing the products of participation. Reification can occur by making, declaring, recasting, perceiving, articulating, designating, representing, using, etc. It can take many forms, for instance, it could be awarding someone a formal title or position, creating an intangible formula, an articulation of a political sentiment in a slogan, a scientific specimen, a piece of code, etc. and the process and practices involved in producing these products. A risk associated with reification is that it can possibly fail to capture the true or intended meaning and trivialize a complex nuanced sentiment with a catchphrase or by categorizing people it may designate a stereotype.

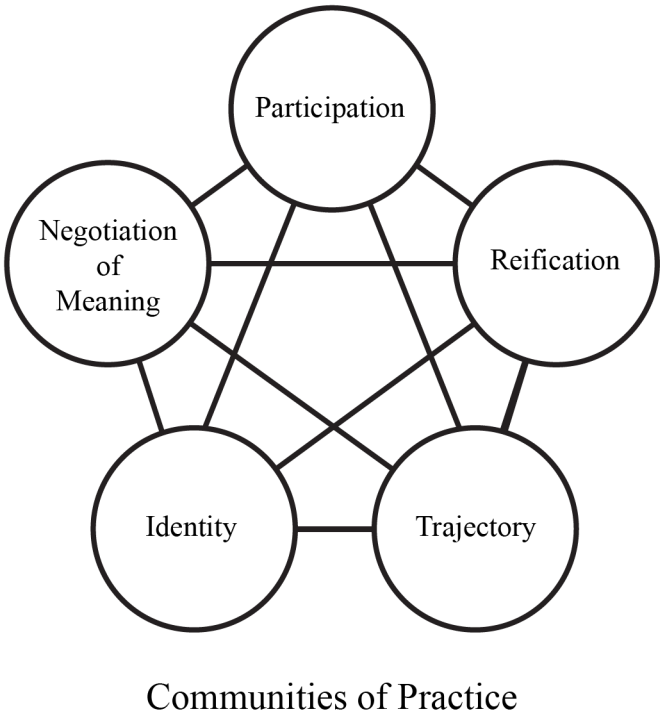

Figure 1.3: Central concepts of Communities of Practice

Participation and reification exist in a duality, where they complement each other as a balance. Participation produces products that are recognized as reification, and together they

allow for the negotiation of meaning. The products alone may be useful but not completely meaningful. For example, a piece of code and the process of writing the code can serve as reification of the programmer's participation in the software development community. Even though the code can be used without the programmer, it serves to legitimize the programmer's efforts and in turn forms her contribution to the community. Conversely, the programmer may be needed to do a code walk-through to explain the code to her peers or even other stakeholders. Similarly, laws can exist stand-alone, but a lawyer or judge may be needed to interpret or represent them to the rest of the community. The title of lead programmer may help reify the programmer's success in the team, but without the programmer occupying that position, the position is meaningless. The balance between reification and participation can sometimes be subtle. Too much reification with little participation may serve to undervalue the participation and the products of participation. However, too little reification when a lot of participation in the community has occurred can discourage and alienate the members of the community.

Wenger also talks about Identity as a combination of the meaning negotiated through participation and reification in a community, as well as the learning trajectory into a community. Wenger argues that Identity is always being shaped and is temporal. It is constructed in social constructs and reflects how one perceives themselves personally and socially as well as how one is perceived in a community. One's trajectory in a community of practice is always in flux. A participant can start as a novice and move into a more central role in the community, and then graduate to vertically successive positions. The participant may eventually feel saturated in interest and contribution in that role and move to a different role in the community or move out of the community altogether. A participant might even determine that they wish to perform a more peripheral role in the community and shape their trajectory accordingly. Figure 1.3 on page 14 depicts the interconnected and interdependent nature of the central concepts of communities of practice.

In software development communities of practice, communication has a profound effect on both meaning and identity. Members are continually and rapidly having to choose between multiple communication options. The right choices at the right time can help bring participants in and facilitate convergence of meaning, while the wrong choices can marginalize participants' identity and confound the negotiation. Although there are many norms and standards of communication that help, members still have to be creative in their context sensitive communication choices on a daily basis. Agile software developmentworks within a framework of communication rituals which also indicates the importance of communication to the software process overall. The rituals and artifacts are not meant to be a manual to resolve all communication issues, they are a loose framework to facilitate better and more creative communication.

Wenger describes how identity is defined by participation as well as non-participation and

how different communities may have different means to reify identity. Some communities may opt for formal and structured reification methods and some might employ more implicit means. Participants in a community also determine their identity by choosing different types of trajectories into the community. Participants thereby contribute to the definition of their identity by choosing a combination of different trajectories, levels of participation and negotiating the meaning of their identity through their quantity and type of contribution.

Considering software development communities as communities of practice is not a novel idea. Paasivaara and Lassenius [50] study "large, distributed agile software development" as a community of practice at Ericsson. Their work describes a large software company designing their processes aware of Wenger's notion of Community of Practice. In Wenger's work, the term Community of Practice is descriptive rather than prescriptive. So even a dysfunctional Community of Practice is a legitimate Community of Practice. It is important to note that even a software development community that does not identify itself with the term Community of Practice is still a community of practice where members join and leave and work together to share and shape knowledge and negotiate their role and the meaning of the system they are building. Even though the term "Community of Practice" is starting to be used prescriptively to describe self-aware communities, reflecting on how to maintain and improve process through member inclusion, software projects are already active communities of practice. They may be very good and effective communities of practice already without associating with the terminology.

1.5 Research Overview

In this dissertation, we investigate the mechanics of how meaning is negotiated in the context of software development. This negotiation is mediated through language and other forms of communication and can occur in different settings and amongst different types of participants. We focus on how identity within the software development community of practice affects participants' communication choices and how in turn, these communication choices can affect others' sense of identity. This effect on others' identity can be either strengthening their connection to their role in the community and encouraging deeper participation and facilitate meaning negotiation or by marginalizing their identity.

We also want to investigate how an individual's notion of identity is affected by their 'trajectory' in the community, taking from Wenger's notion of trajectories. We observe how community members' roles can be determined by their participation and their non-participation and its relationship with the chosen trajectory.

We also want to leverage different pedagogical practices to expose software development students to the complexities of real world software project communication to prepare them for their eventual roles in it.

1.5.1 Research Questions

Our overarching goal is to study how software development communities sustain themselves through change in personnel and priorities and to educate the larger software development community towards their communication choices.

Our objective is to study the way communication choices affect the ability of a software development community of practice to sustain itself. We use the framework of community of practice, we study aspects of the community like identity, trajectory, participation and negotiation of meaning. We study the interplay of these aspects and their relationship with communication choices.

We hypothesize that relationships exist between aspects of a community of practice like identity, trajectory, participation, negotiation of meaning and communication and organizational strategies and patterns.

We articulate our research goals in terms of relationships between aspects of communities of practice:

Question I: Investigate the relationship between choice of communication strategy and negotiation of meaning.

Question II: Investigate the relationship between communication strategy and identity.

We articulate the goal of educating the larger software development community at large towards their communication choices:

Question III: Pedagogical - How can we build awareness of strategic communication among community members like software development students and give them the skills to think of communication as a problem solving?

1.5.1.1 Refined Questions

For questions I and II, we understand that negotiation of meaning, identity, participation, trajectory and reification do not exist in isolation, and are best studied together, affecting each other.

We recombine Questions I and II to reflect the intermixed nature of these aspects of communities of practice.

Ethnographic Goal: Investigate the relationship between communication strategy and the elements of communities of practice (negotiation of meaning, trajectory, identity, participation and reification) within the context of a software development community of practice.

Considering question III, which talks about using what we learn from the software development communities of practice to educate the larger software industry and education community. There are two distinct sections of the larger software development community - software industry practitioners and budding software engineering and computer science students.

Pedagogical Goal: Use our findings from the software development community of practice to build awareness of and develop skills in strategic communication among community members like software development students and industry professionals.

1.5.2 Our approach

To satisfy the combined questions I and II, we start by applying a grounded theory approach, discussed later in Section 1.6.3, on the open source software development email data, focusing on a carefully selected window of observation which contains noteworthy incidents, and our existing student software development project data to identify instances of mentoring and communication choices.

We then apply a grounded theory approach to parse the data collected from the co-located agile software development communities of practice, and identify instances depicting participation, identity, trajectory, reification, negotiation of meaning and communication choices in any combination of subsets and document them, converting them into a format consumable by practitioners.

To satisfy the refined question III, we first consider industry practitioners. We build on a large body of knowledge - existing and sometimes overlapping libraries of scrum patterns and organizational patterns.

We see our contribution to further the knowledge that the software development industry practitioners and experts maintain to share with the larger community about their communication choices.

Considering the audience of software practitioners in training in the form of software engineering and computer science students, we work on adapting the existing computing curriculum to incorporate exercises that sensitize students to their communication choices through exposure to a scaffolding of instances of real world communication examined through the lens of a rubric such as our home baked communication patterns.

1.5.3 Subjects of Study

Our goal is to build and contribute to the body of knowledge allowing us to identify and create patterns of communication. There is no simple, direct answer to what communication characterizes software projects, which makes the grounded theory approach appropriate for this work. We have intentionally chosen different types of subjects to study, to observe what constitutes legitimate software development communication in different settings. We study two main types of software development communities of practice, the open source software development community model and the small industry Agile software development model. We also bring our learnings into the software engineering classroom to teach students to be more conscious of communication choices.

1.5.3.1 Open Source Development Community

One type of community we study is open source software development. We focus on their discussions on collaborative, long-term software projects. The restricted, asynchronous, text only medium of communication presents unique challenges to developers. They are often communicating on high risk issues across different time zones and with different agendas and goals. Some such discussions are better suited to face to face interaction - like troubleshooting while looking at code together. Participants have to find creative substitutes for these practices within their restricted context. Here, we are using discourse analysis techniques to find recurring rhetorical forms in their communication and study the overall structure of the decision-making process.

We observe that in the chosen open source software development communities, for example Audacity development, there is an implicit hierarchy, even though any participant developer, old or new is encouraged to contribute. There is a core "Audacity Team" which often has the final say in major decisions, or can decide what matters need experts to consult. We see examples of community members with an inbound trajectory, newcomers with the intention of full participation, who contribute as a developer over a long period of time and may be finally reified as an official member of the Audacity Team or as an expert contributor in a particular area. Some members choose a peripheral trajectory, where they only intend to participate as an outsider, never as a full participant. These members define their role and identity through their non-participation. They may choose to never participate in decision-making, and confine their contribution to small patches of code. Some participants choose a boundary trajectory, serving as a bridge between the developer community and the quality assurance team or as a bridge between different developer communities like the unix development and the windows development team or the mathematical domain developers and the user interface developers.

Reification and recognition may appear in different forms for members of the community on different trajectories. Some may feel the need for official reification or recognition of their named roles and some might be content with being recognized as a contributor without an official "expert" title.

1.5.3.2 Industry Agile Team

In contrast, we also study a co-located, highly synchronized and structured, Agile team. We focus on a time when the team is transitioning in size and scope after having been ardent and loyal followers of Agile for a few years. They operate within a cohesive team with a strong common vision and a rich range of communication options available to them. They are also a set of developers that have to frequently work together with scientists, engineers and other domain experts, which can present its own challenges. Some of the development teams operate in one location and some other developers work remotely, either from a distant office location or from home. Many of their practices take advantage of face to face interaction and they practice many Agile prescribed communication acts.

Here, we are using semi-structured interviews, observations and capturing the daily experience of software developer through participant observation. We are focusing on how newcomers are "onboarded" or brought into the team and how the communities sustain themselves through change. We take a positivist, quantitative approach of analysis with the developer's daily communication activity data and a we take a qualitative approach for the data collected through interviews and observation.

We observe that the team has a more explicit structure and the hierarchy within the team is identified through formal roles, positions and named responsibilities. An individual's trajectory is more obvious and explicitly discussed. Roles and identity are reified through formal titles and changes in role and position.

Compared to the open source development community of practice, this community has a more structured format for participation and negotiation of role. They also have a formal "onboarding" period and communication rituals attached to it, followed by integration into daily work by pairing with a more experienced programmer. We study mentoring, critique and reflection practices in the different development communities.

1.5.3.3 Software Engineering Student Community

Software projects require sophisticated and varied communication skills, but computer science and software engineering students tend to get little communication training within the discipline.

Another very different type of "onboarding" occurs when we educate software engineering students to prepare them for a professional role in the software community. Undergraduate software engineering students are trying to find their identity in this community. We take our learnings and explore how to help students find their identity as Agile communicators. We bring the discussion on the complexities and nuances of communication in the software project context and the set of communication skills required into the undergraduate CS software engineering curriculum. We do this by bringing in examples of communication from real software projects. We employ a guided enquiry process to assist the students in discovering and analyzing the nuances of these interactions. Then, as the students engage in their own software project experiences, we guide them to reflect on their communication practices. Our goal is to prepare students not just in effective genres of communication but also in sensitizing them to their communication choices.

Our earlier research explored the benefits of using "homegrown" Computer Science student case studies [65] as a tool for students to learn about communication practices in the context of a software project [11] [10].

The third-year Team Software Project course, a requirement for the computer science and software engineering degree programs at our institution, is an ideal venue for communication related instruction to help the students find their place in the software development community of practice. Here, students are exposed to different means of being part of their software community and are given tools to help them engage with their community of

practice better.

Building on two years of experience with programming, software design and computer systems, students take on a semester-long project, with the instructor acting as client. The technical toolset developed in introductory courses is brought to bear on a real software problem. Here is where the notion of software process – the practice of creating software products in a replicable, reliable way – can be addressed and put into action. Techniques for effective communication are obviously an important component of this agenda.

Our Team Software Project course includes an introduction to the concept of software process, focusing on the Scrum framework [55]. One advantage of placing our instruction in this context is that Scrum explicitly acknowledges the importance of repeated, well-constructed communication [64]. Many of the iconic practices of Scrum – stand-up meetings, sprint retrospectives, planning poker – are designed to increase discussion, reflection and debate, all of which help to strengthen the software process. The message that we wish to add is that Scrum, or any other process framework, can provide only broad guidelines for communication, not narrow, comprehensive rules. For instance, team members may follow the practice of daily standup meetings, but it remains to their creative powers to determine what activities follow from the information shared at the standup.

We use a process of guided inquiry [47], where students construct their own interpretations of the subject matter through critical thinking and problem solving. This approach fits the topic well: the search for meaning within a given communication setting is complex, and different observers may see different patterns of communication in play. Guided inquiry allows students to take ownership of their interpretations; at the same time, we consciously steer students away from rote, simplistic answers that ignore the complexity of communication. In the Process Oriented Guided Inquiry Learning (POGIL) [34] model that we adopt, students work in small groups with individual roles: a process framework similar to that of Scrum. The problem solving conversations within the groups give students further practice in team communication.

1.5.4 Data Sources

Our data looks different for the different communities being studied. For the open source software development projects, we have the full account of the email communication in the forum for the windows of time that we have selected. In addition, we have some supplementary data in the form of bug reports, and forum wikis which are sometimes referenced in their email communication.

For the small Agile industry team, we conducted ethnographic study through participant observation and the data collected is in the form of structured and semi-structured participant interviews, an account of the communication events the participant observer attended and reflections of the participant observer concerning "onboarding" and communication style.

For the Team Software Project class, the data collected is in the form of anonymized student submissions for communication centric in-class and homework activities and student feedback on the courses, in addition to our evaluation of student submitted material.

1.6 Tools

We describe the combination of different tools we are using in our research. Depending on the subject, the tools have different degrees of applicability. Our tools help us use tried and tested approaches to guided instruction, conduct research with an evolutionary approach towards refining research questions and to organize our results within an established body of knowledge and add to it.

1.6.1 Cognitive Apprenticeship

Collins [19] outlines cognitive apprenticeship environments where different types of apprenticeship models are implemented to allow knowledge and skill share. Lave talks about situated learning [43] and legitimate peripheral participation [44] where working alongside an established expert, a newcomer to the community learns the practices and acquires the knowledge common to their community. We observe the cognitive apprenticeship model extensively in our study of the agile co-located team. We also employ this model when bringing our communication interlaced curriculum to the software development or computer science classroom.

The basis for our inquiry based curriculum is the POGIL (Process Oriented Guided Inquiry Learning) approach, which originated in undergraduate chemistry education twenty years ago and has been introduced to the computing disciplines with the NSF-funded CS-POGIL initiative. At the heart of POGIL is a guided inquiry learning cycle of exploration, concept invention and application. Students work in small groups with well-defined roles âĂŤ similarly to teams in agile software development âĂŤ to encourage accountability and engagement. Each POGIL assignment has a common structure: supply students with initial

data, guide them through leading questions that allow them to construct a unifying concept explaining the data, then provide means for them to apply and validate their newly constructed concept. It is in essence an application of the scientific method in a carefully crafted classroom setting. In addition to learning the core concepts at the heart of the assignment, students get practice in team problem solving and communication.

1.6.2 Discourse Analysis

Discourse analysis is a blanket term for a different set of analytical techniques based on communication analysis, with a preference for real communication text, rather than simulated or invented communication. Discourse analysis has been used in several fields to study patterns in communication behavior. We adopt some of the principles and practices described in the work of James Paul Gee [30]. Gee's work talks about studying communication for its linguistic aspects as well as ways of participation and representation with its linguistic nuances.

Gee makes a distinction between discourse with a small d to mean any text or speech being studied, but Discourse with a capital D implies both the text being studied and it's context which together forms a way of doing, being or identifying, thinking, liking and perceiving. Discourse analysis thus is not just to study language used but to process any discourse in terms of who, when, what surroundings, what precedes it, how it is intended to be perceived, what it portrays about a way of thinking or being.

It is important to study discourse with respect to the "socially situated identity" of both the creators and receivers of that discourse. What one perceives their identity to be affects the discourse they produce and for whom they produce. Gee also asserts that different, seemingly endless Discourses exist and they are not separated by clean boundaries. Discourses may overlap and interleave and come together to form larger Discourses. A single person may participate and identify with several different Discourses. For example, the larger recognized Discourse of being a software developer may be composed of several different Discourses of novice software developers and more experienced software developers, systems software developers and web developers, database software developers and artificial intelligence developers, user interface developers and animation software developers, etc.

To delve further into Discourses, they can include social language - where different language constructs form part of different Discourses and used with different people. For example, as a software developer, one might use informal language to describe the same issue to a fellow software developer and team member, as they share a common understanding of the system and their tools - the context of the project, but the same software

developers may use more formal and less technical language to describe the same issue to a client or a manager. People engage in different social languages for different audiences, sometimes without even realizing that they do so.

Discourses can also include Conversations - which are often popular social debate which may be about a topic relevant to their Discourse and historically and socially related to their identity. For example, some software developers identify strongly as Apple developers and users, whereas some software developers may identify strongly as a Windows developer and user. These seemingly similar Discourses within the larger software developer Discourse sometimes evoke passionate feelings of being in opposite Discourses, where the software developers might even argue diametrically opposite values, practices and styles. People may own and portray their identity with clothing that indicates through clever inside jokes which camp they belong to and through their participation in different opposing forums where they might voice strong opinions against the other camp.

Another important aspect of Discourses is Intertextuality, where texts from one Discourse may be referred to in another Discourse with the assumption that the recipient of the text have knowledge of the other Discourses. The other Discourses may even be seemingly unrelated, but the participants are aware of the likely link between the different Discourses. As software developers, the obvious common social language would include jargon common to the programming constructs and tools that are used, however it may also include references to different types of gamins Discourses, where it may be assumed that most software developers play certain types of video games or are at least familiar with some common gaming themes and memes.

All these concepts come together in Critical Discourse Analysis where we study the utterance, situated meaning and social practice which together forms the context. It also asserts that the language used both reflects social practices and in that process, it helps create the social context further.

1.6.3 Grounded Theory

Our focus is on communication as a representation of activity in the software development community of practice. We start without knowing in advance what specific aspects of communication would be most appropriate to reveal the kinds of activity in software development that would satisfy our queries. So, instead of starting with a specific hypothesis, we are employing a grounded theory approach, where we start with a generic question and analyze the data in different ways to arrive at the more specific research questions. We adopt the version of Grounded Theory that Strauss and Corbin [58] describe, with three

phases that can overlap.

In the first phase, open coding, that data is tagged using conceptual codes that are developed by studying the data initially. In the second phase, axial coding, the relationship, especially causal, between the codes is explored. In the third phase, selective coding, a subset of the codes and relationships are used to make a theory.

The software development community is vast and diverse, encompassing people from different parts of the world, different academic backgrounds, different skills and communication styles.

The success of a software project, however, is dependent on many factors, one of the most important of which is communication within the software development team and with other stakeholders, compounding the variables in the software process. The same team working on two similar projects with the same timelines and toolsets, may have different outcomes for the two projects.

In the past, we have used ethnographic studies of student software development projects to study patterns and strategies of communication in two almost identical student projects, with common stakeholders and timelines and product, with different outcomes. Using the grounded theory approach there allowed us to develop the notion of communication patterns.

In Computer Science, many aspects of software engineering have been studied using the grounded theory approach. Zieris and Prechelt [70] use the grounded theory approach to study knowledge transfer in pair programming as part of Agile software development, as do Coleman and O'connor [18] in studying software process and software process improvement. Adolph, Hall and Krutchen[32] have used grounded theory to study social interactions in software engineering and [2] share their experiences of using grounded theory in analyzing software engineering practices and communities.

1.6.4 Pattern Languages

Patterns are a tool familiar to the software development community, starting from the very popular Design Patterns [28] to Organizational Patterns [22]. In addition, practices in Scrum have been identified as a pattern language formed of different Scrum Patterns by the Scrum PLOP community [20].

Pattern languages describe relationships between different patterns, even across different types of pattern libraries. Standalone, patterns can describe the players, artifacts, phenomena and practices of software development. Patterns by themselves are even prescriptive limited to a small granularity. However, it is when patterns relate with one another through causal or directing connections, that their extensive descriptive and prescriptive abilities can be exploited. Through their interdependencies and interconnections, pattern languages illustrate the complex web of possible practices and their resulting contexts.

For example, the *Yesterday's Weather* Scrum pattern describes how Scrum development teams can determine how much work they can accomplish within a *Sprint*, using *Estimation Points* from the previous *Sprint*. In this example, the *Sprint* pattern describes an iteration of the structured time period to allow the development with a fixed set of work, the *Sprint Backlog*. The *Estimation Points* describe the point based system of effort estimation associated with a chunk of work, either a *User Story* or a collection of user stories in a *Sprint Backlog* or a *Product Backlog*. However, *Yesterday's Weather* pattern describes a prescriptive and a predictive technique for determining work boundaries for the current iteration of development.

Together, through their interrelations, the pattern languages describe and prescribe the vast variances in practices and outcomes possible within different forms of software development.

Ward Cunningham's WikiWikiWeb, the "original wiki", contains a wealth of named patterns for agile practices, many of which fall into the domain of communication [23]. The Scrum framework is notable in this respect for the way in which it names - and therefore honors - particular communication practices (standup meetings, retrospectives) that would otherwise remain tacit and invisible to students [64].

We have developed the notion of communication patterns [63], designed to capture what is relevant to the software development community about common communication practices and specific communication acts.

The primary way we organize and present our results such that they would be accessible and familiar to the software industry is by building upon the existing and validated body of knowledge comprised in the Scrum Patterns and Organizational patterns work. Our work with Communication Patterns and the aspects of software development that we are studying, fall somewhere in the overlap of Scrum patterns and Organizational patterns. These two sets of pattern libraries are also designed to be compatible with each other, allowing interplay and interrelations between them.

Chapter 2

Initial studies

In this chapter, we present our initial work examining different software development communities and learning about the potential of such data sources and their limitations. The work described in this chapter allowed us to determine the next step in our study - a fully immersive ethnographic study of a software development community of practice. Our initial studies helped us shape our strategy for studying other communities and determine what we capture and how we capture it. As we will discuss in chapter 6, we also use this work to harness examples of authentic, software development communication for use in educating budding software developers.

2.1 Communication Strategies for Mentoring in Software Development Projects [1]

As with professionals in all engineering disciplines, software developers new to a project must be given the implicit and explicit knowledge they need to be productive, in an effective and appropriate way, due to fluid team dynamics, geographical distribution, and other factors. As part of a broader study of communication in software development, we focus here on communication strategies for mentoring. We explore some examples of mentoring-oriented communication, in an educational setting and in an open-source consortium of academics and professionals. We plan to draw out recurring patterns of communication between mentors and protégés.

[1] The material contained in this chapter was previously published in the proceedings of the 44th annual Frontiers in Education (FIE) IEEE Conference, 2014 under the title Communication Strategies for Mentoring in Software Development Projects

2.2 Introduction

In collaborative creative endeavors like software development, newcomers must be brought up to speed not only on matters of fact but on deeper issues of rationale and motivation. The concept of mentor – the experienced guide, conveying knowledge and "know-how" to the protégé – is a time-honored tradition in management. Whether through established, codified practices (e.g. explicit mentoring initiatives by professional engineering organizations [25]) or the more implicit processes captured in legitimate peripheral participation [43], mentors provide instruction, counseling and interaction to impart understanding in a way that "reading the manual" (or the source code) cannot.

Software development, however, occupies a unique position in this space, due to its innately fluid and fast-changing nature. Software teams are formed and reformed at a rapid pace, in response to evolving requirements, business alliances, and personnel changes. Moreover, the flexibility afforded by software development, exemplified most vividly by open-source projects [69], allows theoretically limitless numbers of collaborators, problematizing the notion of team altogether. In this context, the concept of mentor must be expanded beyond its customary definition. Mentoring relationships may be ad hoc and transitory, with little or no clear delineation between those eligible for mentor status and those seeking mentorship. Begel and Simon [8] discuss the importance, advantages and challenges of mentoring for novices in the software industry.

Several scholars have identified communication as a central aspect of the mentoring process. Beyond the "simple exchange of information and accomplishment of ability" which is the primary goal of mentoring, Kalbfleisch [35] likens the process of establishing a mentoring relationship to "the initiation of friendships and love relationships in terms of communicating appropriate relational expectations". Buell [13] expands on this idea by categorizing mentoring relationships in terms of cloning, nurturing, friendship and apprenticeship, and noting the importance of "turning points" where the nature of the mentoring relationship changes [14].

In this section, we explore the communication choices that developers make as they initiate and conduct mentoring activities. Our study samples include a student software development project, with regular face-to-face interactions with a client/mentor, and a globally distributed open source development project that primarily communicates via email. We apply our notion of communication patterns [38] [39] to characterize mentoring activities, employing Buell's mentoring models [13] as a guide.

2.3 Student Mentoring

In this section, we describe our observations of the "Nurturing model" [13] of mentoring and the communication acts and strategies associated with it in two student software projects, where the communication context was characterized by face to face interaction and accessibility to the client/expert/mentor.

Our two student software projects were each a semester long each and consisted of a team of three software engineering students working on a US Navy-sponsored project named "Seabase". The project centered on development of a controller for a ship-mounted crane and involved conversion of some legacy code. The client was "Hank", a professor in the mechanical engineering department who originally developed some of the legacy code. With fresh, inexperienced teams and a short project duration, it is difficult to establish repeatable practices for project work. Students did however have the benefits of physical colocation and a readily available and involved client.

In the two student projects – Seabase I and Seabase II, we witness the "Nurturing Model" [13] of mentor relationship where the mentor facilitates an environment for the protégé to learn and provides help and encouragement with guidance, as opposed to the "Cloning Model" where the mentor issues commands to be followed. So how does the "Nurturing" relationship manifest itself in communication strategies? We see two strategies – an initial Mentor as Interrogator strategy and a more mature Mentor as Oracle strategy, supported by a variety of communication tactics.

Mentor as Interrogator: The classic view of mentor, as illustrated memorably in Socratic dialogue, is mentor as asker of questions, carefully chosen to reveal gaps in knowledge or provoke awareness among protégés. In this pattern, the interrogation is typically followed by advice or sharing of strategies to overcome the identified gap. In his exchange with Seabase II team member "Bob" in Table 2.1 on page 32, Hank is trying to determine Bob's plan for soliciting information from another student team. In fact, for the first few weeks of the project, most of the team's meetings with Hank followed this pattern interspersed with giving advice and taking progress updates. Interestingly, Hank's mentoring in this exchange is encouraging Bob to think strategically about his upcoming communication with the team.

Mentor as Oracle: This is the strategy of learning in the presence of a mentor with the protégé posing questions and the mentor answering them. The Seabase II software engineering team, when tasked with learning MATLAB, spent some time trying to learn from their client and technical expert, Hank's directions and reference material, but solicited Hank's time to ask specific questions about the language and platform where Hank (in the

Table 2.1
Mentor as Interrogator

Seabase II – Week 4

Bob expresses that he might meet a different student team that the SE team need data from

Hank: What do you hope to get out of that meeting?

Bob: See how they are testing code, and how they are using Simulink

Hank: The reason to have that meeting with them was to understand their system, right? Specifically.

Bob: Trying to see what values they are using for Simulink

Hank: Like sensors...

Bob: Yes, like sensors, values for testing

Hank: So what you might want before that meeting is the things you need, like the sensors list that you would need. You won't have a list and they won't have a list, but since you folks have the diagram for that block. Do you know from that what values you will need?

They look at the diagram and discuss some input parameters like sway, swing angle, hoist, lock, etc.

Hank: It might be a good idea to have this picture when you talk to them.

(Bob and Hank together make a list of the values)

Hank: Ok, here is a low-level question. How do you want to go about it when you meet them then? You are at the interface, you provide them the list of things you need. Keep in mind that they are a senior design team just like the platform team. It would be good if you have that dialogue with them. I guess what I am trying to say is that you might not get a quick answer. They should know, but they might not.

role of the mentor) resolved their queries through demonstration.

In Week 7 of Seabase II, Denise, a leading member of the SE team, meets Hank to learn MATLAB, which was a project requirement. After going through the reference material and code examples that Hank shared, the team is having trouble with a specific portion of the code regarding the damping mode block. Hank suggests using one block to calculate the damping values instead. The team watches Hank work on his computer as he demonstrates how some flags are being set in different S-functions of the code. He then tells the team they can choose to use whichever method they like best. Denise then asks about the placement and detection of logical breaks in the code to structure it better and Hank makes suggestions to structure the code better.

We observe a use of the Over The Shoulder Learning [61] tactic in the context of the workplace where the team observes Hank as he works step by step through examples of flag setting in the code. This is easily implemented in the face-to-face synchronous setting available to Seabase II. In Seabase II, we observe an unmistakable "turning point" in Hank's relationship with the team, from the "Nurturing Model" to the "Friendship Model" [13]. In Buell's conception, the Friendship Model is characterized by "collaborative, reciprocal, mutual engagement" and weak or nonexistent hierarchy in the mentor-protégé relationship.

The turning point in Seabase II occurs during a meeting where team member Denise brings an elaborate hand drawn chart to depict data dependencies between blocks of the original legacy code. The chart was something Hank had repeatedly requested of the Seabase I team and finally found to his satisfaction with the Seabase II team. The hand-drawn chart plays a crucial role in demonstrating commitment to the client. Interestingly, the chart originated as a pedagogical tool for Denise, helping her to "get her head around" the legacy code. As such, it is messy and difficult for other readers to understand; however, Denise takes advantage of the synchronous, face-to-face communication with the client to "talk him through" the document, thereby mitigating any confusion caused by its hand-drawn nature. We call this tactic Artifact Facilitated Discussion.

Artifact Facilitated Discussion: This tactic is observed when the presence of an artifact, like a diagram, or piece of code, or design document becomes the center of discussion and facilitates and captures the understanding of the participants. It is associated with communication situations where participants have large gaps in their shared knowledge, where the problem of articulating the question and discovering the right question to ask is difficult.

It is typically found in a synchronous communication setting, so participants can confirm understanding with each other through the "catalyst" of the artifact. This is also an example of incidental learning [21] in the presence of an artifact - the hand drawn chart that Denise

Table 2.2
Artifact facilitated discussion and Mentor as interlocutor

Seabase II – Week 6 – (Denise and Hank are looking at her chart together.) Denise: This is where we need some help. So this is what happens in the code (pointing at Denise's chart) Denise explains on her chart that she has color coded based on which blocks are her responsibility and how the chart describes the blocks. Hank: Can you show me some example within the code? This is great. Don't throw this out. Is this hand-drawn?
Week 9 - (Hank and Denise are looking at the chart and Denise is explaining how init runs and affects other S-functions. Hank asks what some of the functions do, especially init. Denise explains the purpose) Hank: Oh that is sweet! That makes sense now. So when this one is high, that value becomes high and this one goes low, that value is low. I finally get it.. what is setup? (Denise explains what setup is.) Hank: I love it. I love it! The beauty of something like this is that I can understand it. Someone with a high level of knowledge of how the code or the function works can look at it and completely understand it.

made to trace flow of code module dependencies. In this pattern, the presence of the artifact allows for more questions to be asked and promotes collaborative learning. The ability to point at places on the artifact to explain or better ground one's questions is valuable.

Mentor as Interlocutor: Denise's chart initiates a turning point in Hank's relationship with Denise, toward a "Friendship Model" of mentoring. In communication terms, a new strategy emerges – one in which questions arise from both the mentor and protégé and they play off each to share knowledge. Most meetings from the Artifact Facilitated Discussion onwards followed this strategy, where the team, implicitly led by Denise would brainstorm with Hank about strategies for arriving at solutions to identified obstacles.

2.4 Open source mentoring

In our student project case studies, we find that the student teams clearly benefit from the physical colocation of mentor and team and frequent synchronous communication – factors that facilitate more traditional mentoring approaches. How do mentoring strategies change when this easy access to communication is not available? In this section, we describe the mentoring strategies we observed in an open source software project where the communication landscape was drastically different. It allows us to study similar mentoring strategies in contrasting contexts to appreciate the essential attributes of the mentoring strategies that work for different situations.

We are currently studying an open source visualization software project with developers distributed geographically (primarily in Europe and South America) and varying in their levels of experience and of commitment to the project. Communication is conducted almost exclusively on a common list serve. The project has an implicit core group of programmers, who often take on an implied mentoring stance for the "newcomers".

We observe the same mentoring models as the student project – Nurturing and Friendship. The Nurturing model is seen typically between the experienced programmers and the newcomers and the Friendship model exists between the core programmer group. We witness turning points where novice programmers become experts and switch to a mentor role from that of a protégé. However, in this asynchronous medium of communication, we focus on how these mentoring models translate into communication patterns.

We also observe the Mentor as Interrogator and Mentor as Oracle mentoring strategies very often, where when a new or less experienced programmer would pose a question, typically the host would acknowledge it, start with appreciation and encouragement and then pose questions to arrive at the core of the issue. When satisfied that the question is valid and properly articulated, the host would typically answer with a solution along with advice, often with code detail and steps to follow. Email is a less than ideal, asynchronous form of communication and the project members have to use it even for interaction that is typically conducted face to face.

In the student projects, an important recurring communication strategy was the *Artifact Facilitated Discussion*, which is impossible in the open source project as the project members are geographically distributed and spread across different time zones. We examine how they cope to still facilitate incidental learning. The participants are typically proficient programmers, many of whom are well versed with the library they are working on with many years of experience, code as part of the email body very frequently becomes part of the conversation. On a closer, more qualitative look, we observe the *Code As Conversation*

Table 2.3
Code as Conversation

("Novice" replies with corrections) thank you for the navigation. There is the script: <script code pasted in email> ("Host" appreciates the script) Thanks! Now, it will be easier to review :) ("Host" critiques and guides "novice" gently towards other solutions) Hi, I looked at it a bit. It's a start, but I think the direction is not quite right yet. Let's take a look at one of your examples: <code example here> I like that you are using a matrix model. But, what is not so clean is mixing shapes and elements. Right now, you are creating elements within the definition of the shape (i.e., instVarNames). A rule of thumb should be that shapes should be interchangeable. Consider the following Mandorian example: <code example here>

pattern, where participants on the forum ask a question related to the code and paste a code snippet in their email. In turn, respondents also use code in their email to share or propose solutions, along with some text as explanation. This exchange is often used to arrive at implementation strategies, make design decisions or even to debug code together.

In Table 2.3 on page 36, we observe a combination of the Code As Conversation pattern and the Mentor as Oracle and Interrogator mentoring strategies. We share an excerpt from an email thread started by a programmer with comparably less experience on the project

than some of the senior team members. The “novice” programmer wants the “host” to review his code implementation. The “host” starts with encouragement and shares the correct procedure for collaboration. When the “novice” creates the correct script, the “host” programmer informs him that he will examine his code, he soon replies with some comments – he talks to him about the direction of the solution and uses the novice’s code example to illustrate what should be different. He then shares his own code example to demonstrate how to accomplish what the “novice” was attempting.

Although the mentoring models observed in both the student and open source projects have similarities, the change of medium to email only affects the tactics. When tone is not easily conveyed and the ability to point at a collectively viewed artifact is missing, we see that both the “host” and “novice” carefully craft responses with lots of information included, sometimes step by step directions to overcome the lack of a face to face interaction, where even partially articulated questions accompanied with gestures and pointing convey one’s meaning. Pointing at common code is replaced by copying and pasting code fragments. We see that “Over The Shoulder” learning is not possible but “incidental learning” assisted by the code fragments takes place.

Finally, we note that protégés eventually become mentors as an example of the “turning point” where we see a “novice” programmer used to pose questions to the forum very frequently accompanied by statements like “I am very confused” and “I am sure this is a stupid question”. Over the years, the “novice” has turned into an “expert” where we notice him responding to and encouraging “novices” with statements like “It is great that you are working on ... and let us know if you need any help”.

2.5 Discussion

We found that instances of mentoring were easy to identify in the student project and the open source community materials, as for the student projects we had detailed ethnographic ‘fly on the wall’ records and with open source projects, we had records of their email conversations. However, with both these data sources, we found that we were missing a context, that perhaps only interviews of greater immersion into the communities of practice would provide.

We take our basis of understanding of mentoring ‘in situ’ from the student projects and the open source software communities, and employ them in studying and understanding a more involved software development community of practice with greater detail and more immersion over a prolonged period, to better understand the nuances of communication and mentoring within the community, especially the practices that allow these typically heavy

churn communities to sustain themselves.

2.6 Conclusion

From these initial studies, we learn that mentoring in software development communities can exist in a multitude of forms and is an integral part of the community practice. We also learn that without reliable and extensive access to the software development community being studied, we can only get partial interpretations of the stories of these communities. Using, static data snapshots from the communities, a lot of what we understand may be through inaccurate extrapolations. These studies help us determine that our next step is to study a community with greater access, in a more immersive way, to learn about its stories.

Chapter 3

TAI - Participant observation

In this chapter, we describe our methodology and process for conducting ethnographic study of our selected software development community of practice at TAI(name changed to protect its identity), through participant observation. We also share our challenges and limitations in this endeavor. For the sake of the study, we will refer to our participant observer as Natalie where the TAI study is concerned. The names of all the subjects of study at TAI - developers and other roles, have been changed.

3.1 TAI: An evolving development practice [1]

TAI was formed in 1996 as a spinoff from Michigan Technological University, in Michigan's Upper Peninsula. The domain area of the firm — thermal modeling for engineering applications — has remained constant through its history. TAI's identity, however, has changed as it has grown in scale and scope. Market forces have shifted the product vision and the focus of activity. Successful products have led to an increase in scale and therefore a change in makeup and identity of the employees. (Figure 3.1 on page 40 gives an indication of the increased complexity of the code base.) This in turn has led to adoption of new practices and approaches to process.

[1] The material contained in this chapter is a modified version of material accepted for publication in the proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), 2016 under the title Among the agilists: Participant observation in a rapidly evolving workspace.

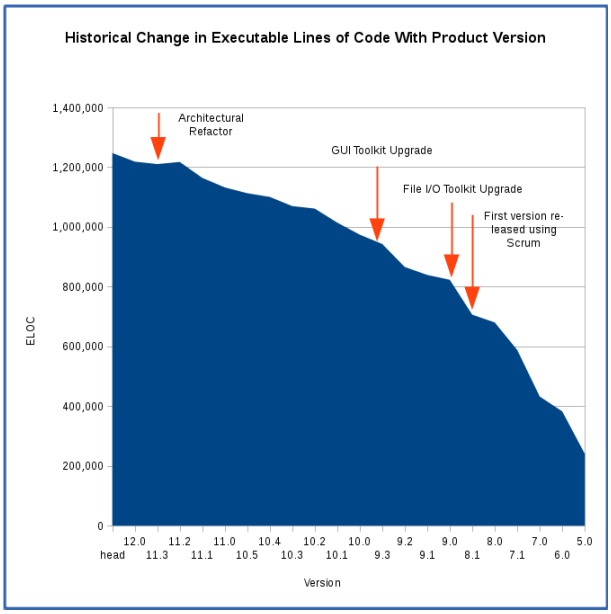

Figure 3.1: Change in lines of code with version number and major events. Code versions in descending order along the x-axis.

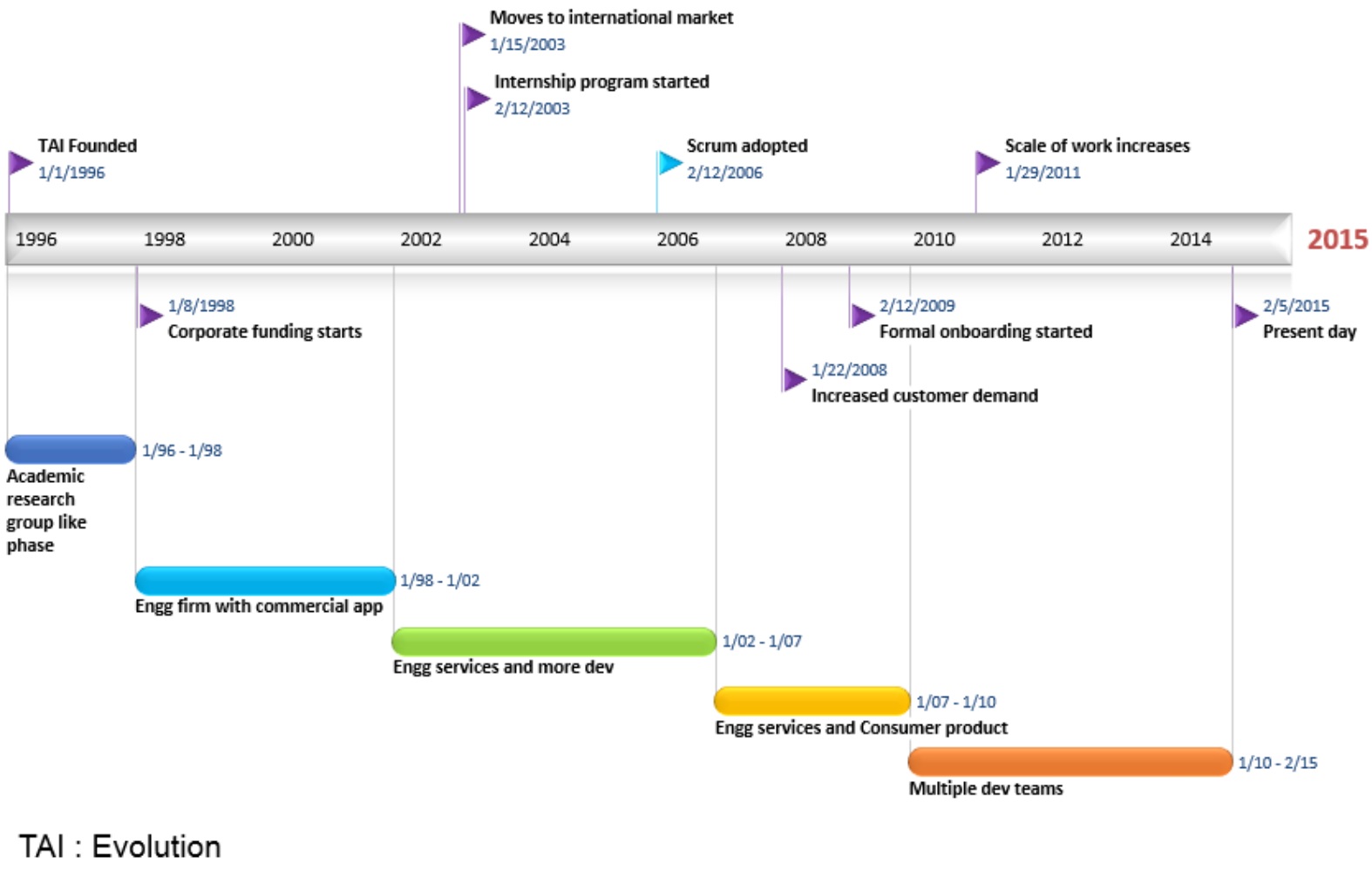

Figure 3.2: Phases of TAI evolution

The rough timeline below and represented in figure 3.2 on page 41 gives a sense of the interplay between these forces.

1996–1998: The firm operates in a way similar to its roots as an academic research group. Funding is primarily from government contracts, leveraging SBIR (Small Business Innovation Research) awards. Researchers work essentially in an autonomous fashion on the shared code base, interacting only occasionally through ad hoc code reviews.

1998–2002: Corporate funding leads the firm in a more targeted direction. With a specific need for a commercial radiation transfer analysis solution, the first commercial application is born. At this stage, TAI still identifies as an engineering firm, not a software firm. Developers have backgrounds in science and mechanical engineering rather than software engineering. Basic software development mechanisms (e.g. revision control, bug tracking, formal code reviews) are brought in, prompted by commercial development. Development, however, still proceeds in a largely decoupled fashion among researcher/developers. There is still no explicit acknowledgement that a principled software process is needed, though individuals are starting to recognize this.

2003–2007: The firm moves to an international market. It positions itself as providing engineering services for hire: TAI employees are hired to model customer engineering scenarios, using the internally developed and used software product. This period is marked by an increase in development, with more people brought on due to external funding. An internship program is started in 2003, and dedicated software developers (not engineers or researchers) are added. Three of the eight developers have CS degrees. Mentoring of new developers is done in an implicit fashion by "pioneers" who did the original exploration of the space. With the increase in scale, addressing issues of personality fit and matching skills to needs proves more difficult. With a new constituency of developers not steeped in the history of the company or the research, further guidance is needed. In 2006 Scrum is adopted as a development framework.

2008–2010: Customer demand exceeds the engineering service model, and the firm pivots toward providing a consumer product, to be offered side by side with service. User experience issues take on greater importance. (Before, dual use applications - use in military and transfer to other domains). In 2009 a formal onboarding process is introduced for development interns.

2011–present: As scale increases, special needs and cross-cutting concerns are identified. The time scale implicit in Scrum is considered too long for basic maintenance — "trying to tango to dubstep", in the words of Philip — so in 2013, a team dedicated to maintenance is formed, using Kanban with weekly cycles. A challenge emerges: how to balance the time commitments of the pioneers between development and mentoring? These team members

have deep knowledge of the code base and can make valuable contributions to the code, yet that knowledge also makes them the best choice for mentoring newcomers. New employees must now rely on non-pioneers for their onboarding experience.

The engineering services and research department members or the product owners often serve as proxy clients. Depending on the strength of the software development department, big stories like feature requests and major bug changes are taken up by the 4-8 person teams following Scrum, where the team commits to a set of stories for a 3 week sprint cycle. Depending on the size of the story, an appropriate number of developers self select to work on a story.

With 20 developers now on the team, Philip finds that he cannot handle all as direct reports. In his account of the early days, "you had to know everything about everything". Now with 2 million lines of code, this is not feasible. He notes a struggle with Scrum: while avoidance of narrow silos is advantageous, how general do you have to be? For real employee satisfaction, there must be a certain catering to personal interests and a vision of ongoing professional development.

The decision is made to set up skill teams with domain areas of expertise. Organizational patterns using the Spotify model [37] are adopted. Vertically, squads work as cross functional teams on future feature sets. Horizontally, chapters are formed on the basis of cross-cutting subject matter expertise. For instance, the physics chapter includes more science and engineering focused personnel, interested in numerical methods; the presentation chapter deals with user facing issues and technical documentation; and the productivity group discusses process issues: testing, DevOps and agile. Each chapter has a leader with traditional management responsibilities. Chapter leads determine who (among those who have expressed interest and commitment) can join a chapter. Philip is mentoring chapter leads. The mentoring process for chapter members is a work in progress.

Philip relates his account of the current onboarding process: small groups (one on one or two on one) work on a specific project for 2–4 weeks, writing a utility program or an exploratory feature, or working on maintenance of the product line, fixing defects. Onboarding is experiential learning, with indoctrination into process woven into the technical work. During the first week, newcomers read up on Scrum and coding standards. With a customer proxy and their mentor, they define stories, then requirements, acceptance criteria, design solution, and implement/test. Newcomers engage in pair programming, two screens, one for code writer, other for test writer. Reflecting on his own mentor role, Philip describes a sensitivity to personality and skill level, and deliberately "driving" less as he senses greater confidence on the part of the learner. Mentorship is a rotating responsibility: mentors are called off of their other duties for extended periods: multiple weeks or months. Philip assesses potential mentors for fit to the work: "we wouldn't do that to every person".

3.2 Methodology

One member of our research team worked as a software developer in the industry software development team for eight months, studying the team as a participant observer and conducting occasional interviews with key team members, especially those that work closely and have varying degrees of experience and serve different roles. In preparation, our participant attended a graduate level course on ethnographic methods, and attempted to derive what subset of methods would be most useful for our particular situation.

In studying TAI, we opted for a mix of immersive ethnographic field study over a long term observation window, with participant observation and informed interviews [60]

3.2.1 Types of data collection

Our goal was to study the software development community at TAI using a mix of both quantitative and qualitative methods. The quantitative data give us insights into the kind of communication activities developers typically engage in, under different software methodologies, over their initiation time and during regular operation. The qualitative data help answer our questions about events and motivations of the developers based on their individual journeys, trajectories and level of experience.

Our plan was to capture three types of data: 1) the participant observer's daily communication activity data for questions like how often do developers work together or alone and for what kind of work; 2) personal reflections of our participant observer about the communication and teaming practices regarding onboarding, role and knowledge sharing; 3) interviews of selected team members to capture their experiences and journeys.

The participant observer tracked the daily communication data by using a calendar and adding entries for the time blocks that they worked in different modes: *e.g.,* a different time block for when Natalie was working alone on a task, a separate time block for when Natalie paired with another developer for the same task, a separate time block for when Natalie attended the Daily Stand Up meeting, a separate time block for a design discussion with two other members. This calendar data was then converted into a format more appropriate for analysis.

3.2.2 Participant observer trajectory

Our participant observer started working at the company as a software developer intern in January of 2015. It was the first day for two other software developers as well. They were given a tour of the offices and were introduced to other software development team members. They had a short session with the HR, followed by an IT introduction, getting access to their computers, software and the common code base. The three developers are then given their team assignments.

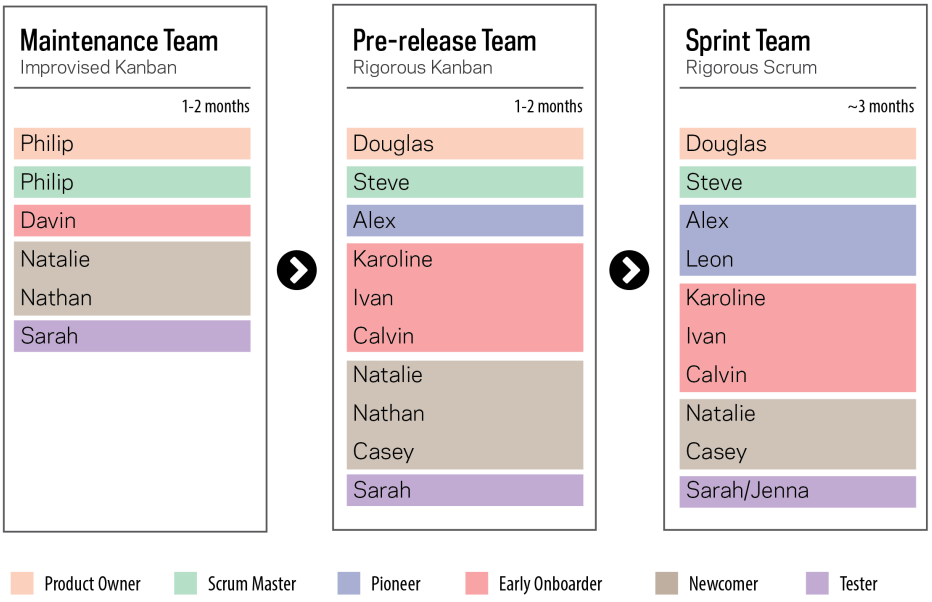

Figure 3.3: Natalie's team membership

Figure 3.3 on page 45 shows the teams that Natalie was a member of, the duration of her stint in each team and her team members. Our participant observer Natalie was placed in the maintenance team along with Nathan. The maintenance team followed Kanban and now consists of one experienced member, Davin and the two newcomers - Natalie and Nathan. As part of their regular process, it is not unusual for developers to move between the Scrum development and the Kanban maintenance teams. The maintenance team at TAI has been transient in nature. New developers and interns are often placed into the maintenance team when they first join. After a few months or so, they are moved to the Scrum development teams and other developers are brought into the maintenance team. There are times when the maintenance team is dissolved altogether and the functions performed by the team are distributed among other members.

At the time that Natalie and another newcomer developer joined the maintenance team,

there was one other developer working in the team, who helped onboard Natalie and the other developer. They held a stand up meeting every morning, but it was more for the benefit of the department manager. The team members sat close to each other and worked on most bugs as a pair. They communicated with each other multiple times a day and as the team was small, everyone knew what work was being done. However, as two of the three team members were still learning how to do the maintenance work, less work was being done overall than what a fully functional three developer team can manage. Some primary features of Kanban like the work in progress limit were not used, as a scenario to use them did not come up. There were times during the next month and a half, when the team grew to four and shrank to one breifly.

After one and a half months of working on maintenance as a three-person team, three to five new members previously working on the development team were added to the maintenance team and the work was re-prioritized to bug squash for a major version product release. This team was rebranded the “pre-release" team. In a larger team, a more authentic Kanban practice was possible. In this larger team, more rigorous Kanban practices were adopted, like revisiting the work in progress limit periodically based on team size and availability. This was also a time when Natalie and the other new developer had learned how to handle the development work and were able to contribute meaningfully. A more rigorous template for bug resolution reporting was followed and it was determined that there should be a formal requirements gathering discussion and demonstration of the resolution to client for all bugs. This was not something the maintenance team had been doing for the month or so before that, but when other developers had been on the maintenance team a year before that time, these were common practice. Over the month and more, the pre-release team also grew and shrank in size periodically, depending on availability of developers and need on other projects.

After working in the maintenance team for a total of about three months, Natalie and Nathan were assigned to one of the scrum development teams, where a lot of the other pre-release team members were also assigned. They worked on Scrum style development for the next three months. Even in the Scrum teams, the team composition changed from time to time. In the scrum teams, practices like daily stand up, estimating, retrospective, etc. were adopted. During this time, a new software departmental structure was being planned and started to be implemented in small waves, where reporting and expertise groupings are changed.

3.2.3 Selecting candidates

For the interview data, our approach was adaptive. Initially, we were just observing by listening and casually asking questions about the different developers and roles, to identify suitable candidates. We decided to employ informal interviews[53] which are typically used as an early observation tool to gain a working understanding of a new social setting. Informal interviews are typically unscheduled and conversational, not following a script.

In the first few days, as part of her introduction to the people working in the company, Natalie let them know that she was working as an intern to serve a selfish purpose of studying the software development at the company for her research. She was also gauging who seemed more likely to talk about it at a later time. She remembered to engage people as she worked with them. A convenient time to engage them was when an open conversation would start about some polarizing topic that occasionally many people from the software development department would participate in. After such a conversation would come to its natural close, it would be a time to ask people if they wanted to have lunch together with a larger group. Many people would eat by themselves at their desk or in the common break room if not asked otherwise. It was prudent to engage developers and other roles at lunch about how long they have worked for the company, what their background has been, whether they have any views on software methodology, etc. There were some developers who were aware of our intentions before we started working at the company, and expressed a lot of enthusiasm for supporting the study. Some of these developers were the first few that were considered viable candidates for the study.

Working in agile teams, where the teams are self-managing, and all developers are supposedly peers, and different teams all report to one department manager made it challenging to identify what different implicit roles the developers were playing. There wasn't a traditional corporate ladder where people at different rungs had different positions. Conversation and observation were the tools we had to determine who does what in the teams. Initially, while working on a new bug, Natalie would notice whose name would be recommended often for getting advice or help. She would also notice whose advice was most useful and seemed most insightful. She would also notice which developers would share advice and an explanation for that choice. Something Natalie was surprised to find was that confidence was not a good predictor of actual knowledge. Many young and inexperienced developers would also very confidently give advice that may not work and very experienced developers might give great advice but might not appear as confident when they do.

Natalie would notice when a developer's name would be touted as a good source of information on a variety of subjects and would also notice whom other, more experienced

developers would go to for advice. Natalie also noticed the variation in opinions and adherence to rigorous process. This was a criterion in selecting different candidates who seem similar in experience and knowledge but follow process differently.

After a few weeks of getting to know the different developers in different teams, and getting to a place where Natalie was comfortable with the work herself, she spoke to the gatekeeper for the study, who is also the head of the development department and has worked in the company for a long time. She ran the list of proposed candidates for the study by the gatekeeper, and he recommended more names based on their different trajectories and working style. From that list of names, some of the candidates were approached. We intentionally picked developers with vastly different levels of experience, ranging from someone who has been a developer for decades, to developers who have a few years of experience as part of this company, to developers who are in the process of being inducted into the community during our observation window. Not all the potential candidates identified through discussions with the stakeholders were included in the study. Some candidates were willing to participate in the study but were largely unavailable due to time conflicts and work related travel and vacation time.

Our final small set of developers were selected also based on the likelihood of their working regularly on the same team with our participant observer, thereby allowing us to study them in their normal business day and also study them through the interviews. It would also be more convenient to find interview time with developers whose daily work schedules are being closely monitored by us, as part of our daily operation.

3.2.4 Asking the question

The initial question of asking someone to be part of the study was tricky, as it meant getting time alone with a developer while you are not talking about development work. For the sake of privacy, the question of participating in the study needed to be asked alone and all subsequent interviews needed to be conducted alone as well. The way the developers are seated, it is rare that one would be out of earshot of everyone.

In preparation for asking developer candidates to agree to participating in the study, we had to anticipate what their concerns would be. Their primary concern would be time and availability, which is something we would have to work around. Their other concerns, potentially about being judged or misrepresenting the company, were far easier to allay.

Natalie's preferred time for asking developers to participate in my study was in the later half of the day, or just after lunchtime. These would be times when developers would take

a coffee break and it would be possible to catch them alone as they walked towards the break room. Once we got past navigating when to ask developers, the actual 'asking' was relatively simple. Natalie would start with a reminder and brief description of her study and ask if they would be willing to participate. She would describe what their participation would entail, stressing on how it would not get in the way of their work and we would work around their time. She would also make it clear that they are not being judged and neither is the company, and that we are interested in studying software development. She would tell them that her intention is to occasionally conduct about five to fifteen minute interviews, based on a convenient time for them and to let them know that they could end their participation whenever they wanted.

3.2.5 The first interview

The interviews were conducted with some planned seed questions and more probing specific questions constructed 'in situ,' meant to guide the interviewee to elaborate on their story. For the regular, more in-depth interviews, we initially considered a semistructured interview approach [33] [45] where for the most part, a script of pre-determined open ended questions is used, with careful ordering and phrasing for later comparability. Semistructured interviews are more suitable for studies where multiple interviewers study many different subjects and the interviewers do not have a lot of knowledge about the subject's social context. We also considered the unstructured interview approach [53], where formal, scheduled interviews follow more of a conversation format than an interview where the nature of the questions can be open ended, and it is encouraged to ask follow up questions based on the answers received. As our goals were to capture the participant's stories, journeys and motivations and we were less concerned with comparability the open ended questions served our purpose. We adopted a mix of semistructured and unstructured interviews, leaning more towards the semistructured format described by Leech [45]. Like the semistructured interview approach, our interviews depended on forming a rapport with the interviewees, asking grand tour questions like "Describe your onboarding experience at the company." and prompts when there is more to explore on the current topic, even if the pre-determined question has been answered. For the common questions about background and trajectory, the format and order of the grand tour questions was kept consistent, but the follow up prompt questions differed based on the interviewees' answers.

Our judgment sample of interviewees was also small, and our participant observer was fully immersed into the field's context, which made unstructured interviews ideal.

The first interview for each of the participants was rather long, compared to what we had indicated we would normally want to engage their time for. It involved explaining the user

consent form and again explaining all aspects of what they were agreeing to and their rights. This took about five to fifteen minutes on its own. In some cases, the user consent form was signed on some day before the first interview. It was also important to tell participants that they did not have to answer based on what they think we would want to hear, that their actual experience is more valuable.

Then we finally started with the interview. For most participants, we started by asking them about their background before they joined the company and their initial experience as part of that company. We had to be cognizant of time, even though the participants seemed engaged and sharing well, to not discourage their future participation based on how the first interview was longer than what we asked for.

From the first round of interviews about the developers journey into the software development community, we realized that many participants credit, in part the different forms of help or guidance they received from other developers. This guided our subsequent questions at later interviews.

3.2.6 The Routine

After the first interview, a lot of the participants realized that the interviews are fairly simple, albeit longer than they may have planned for. However, finding time for subsequent interviews proved to be tricky. We kept in mind how the end of the sprint tends to be busier and leaves developers with less time to spare. We aimed for earlier in the sprint to ask for interviews, and continued favoring the later half of the day. It usually helped to ask the candidate in the morning if they would have time for a five minute interview later in the day. It also helped to have most of the questions determined beforehand. This helped participants feel like they are not wasting their time during the interviews. However, during the interviews, it was useful to pursue an answer with a follow up question, even if it was off script. Especially if it was a part of a story that the participant appeared excited abut sharing. It was sensible to allow the participants feel heard, even if that line of questioning did not directly relate to our focus.

We would also follow up with more questions whenever the participant would talk about their identity in the workplace, or their role or relationship with others, especially how these may have evolved over time. We found that the more experience the participants had, the more comfortable they seemed and the more verbose their answers were.

3.3 Challenges of the ethnographic process

Through this ethnographic study, we did not expect to capture everything that defines the software development community of practice at TAI. It is a community that has evolved over a long time to its current state and is still continuously evolving. The community has grown through different phases, priorities and skill sets, employing different practices and models of software process to find the combination that works for them and then consciously improving upon it. This dynamism made the community more attractive as a subject of ethnographic study, but also posed challenges. As the core concepts of community of practice like participation, reification and identity are different for everyone, the community in general has different meaning to its participants. We attempted to capture what the community means to some of its selected participants through their experience of it.

3.3.1 Dynamic teams

For a small development firm, it was a very dynamic workplace in terms of team composition, nature of work and roles, which posed some challenges in studying it. The pre-release and the scrum development teams had a mix of developers with variable levels of experience. What we found was that they had all carved their own niche and developed their areas of specialization, based on their experience, interest and abilities. However, these roles were mostly implicit and there was no place where they were formally described. It was only through a combination observation and multiple rounds of strategic interviews that we were able to learn about them.

As the developers have been working in the development team for a while, some of them had to think about how to articulate the things they do and the implicit roles they play, as they might be taking it for granted themselves.

One of the developers mentioned how even within the small set of less than twenty active developers, when teams are formed, different teams end up having different dynamics. There have been instances where certain teams were intentionally changed in their composition, as they found different combinations of developers to work more effectively or happily together. Sometimes, the implicit roles involve more than just knowledge sharing responsibilities. Some implicit responsibilities could include playing an arbitrator when small disputes in the self-management of the team may occur, or to serve as a calming, equalizing force when the team is tending towards discord. Karoline reports of a case

when Oswald's tone in correcting other members of the team was causing different members to feel disparaged, especially considering that Oswald was the youngest team member. After Gerome, who is older but has comparative experience on the code base, was added to the team, they found that Oswald's tone was not leading to minor disputes as often within the team.

These were aspects of the implicit roles that would not be possible to discover at all without open ended interviews and observation.

3.3.2 Company culture

As part of a rigorous agile workplace, there is emphasis on extracting a lot of value from the time spent at work. This manifests itself as an explicit and an implicit focus. This time based efficiency is deeply ingrained in the company culture as well. There are a few aspects of the company culture that make the workplace very effective but also made it difficult to find time to talk about things other than work. It is heavily encouraged that everyone only work for forty hours a week, and no longer. Many developers come in early in the morning to get some time alone to work and then leave when their day's work is done, sometimes even in the late afternoon time. Most developers are encouraged to not work late often and are not allowed to take their work home, unless they are among the few remotely working developers.

Perhaps because of their government contract work background, the company requires developers to document how they spend their time, for the interest of bookkeeping and billing, down to the granularity of minutes. This means that it is rare for developers to spend time on anything other than work and even more rare for developers to linger at work after their hours in the week are done or if the work for a day is done. So asking anyone to participate in the study or have to give an interview meant that we are asking them to spend their time on work that they are not getting paid for and work that is probably delaying their return home for that day.

The workplace where the developers sit is usually a quiet space, where most people have their headphones on while working. Another reason the workplace is so quiet is because some researchers sit near the developers' workplace and are disturbed if loud conversations take place. There are times when developers start an impromptu conversation and others chime in with their opinions. These are usually followed by a complaint from the researchers sitting nearby. Most conversations occur between two to three people in preferably soft voices or inside the meeting room.

3.3.3 Multiple identities

During the interviews, it was important to not let our personal opinions about people guide our questions. It was important to be cognizant of not allowing the interviews to drastically change our relationship with the participants. It was also important to not let the interviews affect our observations during our daily work. It was important to remain cognizant of our different identities - developer, observer, interviewer, recorder of interviews and daily activities. There was another identity - that of a friend, colleague and confidante.

There was occasional conflict in the multiple identities. When an aspect of the communication was shared as part of a formal interview, where one or the other developers were not presented entirely positively, as a participant observer, it was important to attempt to remain unbiased and report faithfully, however, as a friend, it was difficult to not put a positive spin and occlude the facts.

When unfavorable opinions about other developers were shared, it was difficult to not feel like reporting on them is somehow a betrayal of trust from a confidante as our participant observer was a confidante to both the reporting party and the subject of the opinion. Similarly, it was important to not skew a story to make your friend the hero, when something positive is revealed.

As an interviewer who is also a colleague, it was important to not let our choice for a daily lunch companion or teammate be affected by the backgrounds and histories revealed during the interviews.

Chapter 4

TAI - Data and Discussion

4.1 Evolving meaning, evolving identity

We interpret communication in software projects as a negotiation of meaning. Wenger [67] captures this negotiated quality of meaning in his concept of Community of Practice. Wenger describes Communities of Practice as "groups of people who share a concern or a passion for something they do and learn how to do it better as they interact regularly". He views meaning as essentially linked with identity within a practitioner community. Those who identify as community members contribute to the shared negotiation of meaning, and participation in the negotiation of meaning reinforces identity.

In computing disciplines it may seem unusual to think of meaning as negotiated, as its roots in mathematics and use of programming languages and algorithms makes us think of meaning as precise with little ambiguity in behavior, as with code. However, in software engineering, code is only a manifestation of the developers' or designers' mental model of the system and that model in turn is a representation of users' and clients' requirements. To arrive at the precise code, interaction is needed among multiple stakeholders with different roles and involvement in the software development community or that particular software project community, and the "meaning" is negotiated through different and possibly conflicting interpretations of the requirements and design specifications. In addition, a community member's role, influence and contribution in a software development community of practice may also be negotiated over time as members join and leave and work in different capacities over time, leaving the community in flux.

When one thinks of software development in an abstract form, one thinks of developers as

a homogeneous group of skilled and experienced programmers following predetermined processes to deliver a product, however in practice, the developer community is always in flux as new members are learning the ropes and experienced members are taking on larger tasks or moving on to different roles. In addition, the meaning of software is always changing based on changing requirements and resource limitations.

At TAI, the meaning of onboarding and mentoring has evolved. This is partly due to market demands — as their software becomes a full fledged consumer product, and researchers no longer have the luxury of total independence they once had. It is also due to scale, as new developers outside the area of expertise need to come up to speed. The evolving meaning is partly articulated through explicit changes from management (Karoline and others), but as we shall see, it also occurs in a tacit manner, as newcomers improvise new activities and relationships to get what they need.

Examples of negotiated meaning in the TAI case (which we discuss at length later):

† The maintenance team becomes a pre-release team and increases in size from three to eight, with a switch from Scrum to Kanban. The larger team, is a mixture of newbies and more experienced employees. How does the new way of doing maintenance get defined?

† Within the maintenance team, a discussion emerges concerning the template for closing a bug. What is the appropriate level of rigor for such a template?

† In the maintenance retrospective, it emerges that some bugs require work that exceeds the bounds of Kanban. Should bugs be sized as stories? Is the Product Owner the appropriate arbiter for bug size?

Examples of evolving identity at TAI:

† Pioneers find themselves in a difficult situation: with their intimate knowledge of physics and code, they are extraordinarily productive in terms of story completion. But ultimately, they have more value in serving as mentors and educating others. Later we focus on Alex's journey from researcher to programmer to implicit then explicit enabler.

† Similarly, employees who were onboarded in an earlier era, receiving their mentoring from pioneers, are now put into role of mentor. Karoline, who experienced different mentoring styles as a protégé, reflects on that difference in producing her own style when she becomes mentor.

4.2 Mentoring

In collaborative creative endeavors like software development, newcomers must be brought up to speed not only on matters of fact but on deeper issues of rationale and motivation. The concept of mentor — the experienced guide, conveying knowledge and "know-how" to the protégé - is a time-honored tradition in management. Whether through established, codified practices (e.g. explicit mentoring initiatives by professional engineering organizations [1]) or the more implicit processes captured in legitimate peripheral participation [44], mentors provide instruction, counseling and interaction to impart understanding in a way that "reading the manual" (or the source code) cannot.

Software development, however, occupies a unique position in this space, due to its innately fluid and fast-changing nature. Software teams are formed and reformed at a rapid pace, in response to evolving requirements, business alliances, and personnel changes. Moreover, the flexibility afforded by software development, exemplified most vividly by open-source projects [69], allows theoretically limitless numbers of collaborators, problematizing the notion of team altogether. In this context, the concept of mentor must be expanded beyond its customary definition. Mentoring relationships may be ad hoc and transitory, with little or no clear delineation between those eligible for mentor status and those seeking mentorship.

Several scholars have identified communication as a central aspect of the mentoring process. Beyond the "simple exchange of information and accomplishment of ability" which is the primary goal of mentoring, Kalbfleisch [35] likens the process of establishing a mentoring relationship to "the initiation of friendships and love relationships in terms of communicating appropriate relational expectations". Buell [13] expands on this idea by categorizing mentoring relationships in terms of cloning, nurturing, friendship and apprenticeship, and noting the importance of "turning points" where the nature of the mentoring relationship changes [13].

4.3 Qualitative Data Analysis

4.3.1 Stories

We selected some participants of the software development community of practice, to study in greater detail and inspected their trajectories through the community, their experience of

Table 4.1
Selected people at TAI

Characteristics	Alex	Karoline	Ivan	Casey	Philip
Formal education	Mechanical Engineering	Computer Engineering	Computer Science	Computer Science	Mechanical Engineering
Years of experience prior to TAI	0	2 yrs	>10 yrs	0	>5 yrs
Years at TAI	>15 yrs	>2 yrs	>2 yrs	<1 yr	>10 yrs
Starting role	Software developer intern	Software developer intern	Software developer	Software developer intern	Software developer
Current role	Senior software developer	Software developer	Software developer	Software developer intern	Software Department Head
Mentoring role	Pioneer	Early onboarder	Early onboarder	Newcomer	Pioneer

onboarding, their identity and its evolution over time, their participation and how it has changed over time. We list them in table Table 4.1 on page 58 which describes their basic differences in experience and background. We briefly describe their stories here and discuss aspects of their stories in subsequent sections.

4.3.1.1 Alex

Alex started as double major with mechanical engineering as one of his degrees. He identified as a mechanical engineer when he first started interning at TAI to work with Leon, who had recommended him for the position. This was very early in the inception of TAI as a company, and business still operated a lot like a research group. The main portion of

the program was Leon's thesis, and Alex was assigned to work on a GUI for it. He was told to design and build the GUI as a single project, which was a mammoth task in itself. At that time, the internship model at TAI was such that interns did not have to attend work all day. Alex would work for a few hours at a time based on what his coursework schedule and other priorities allowed.

He then left TAI and went to work on the west coast as a drafter for a couple of years. At that point, the TAI founders offered him a job back in Michigan and Alex accepted and joined TAI as a full-time employee.

Reification and identity: Even though Alex identified himself as a mechanical engineer who programs, when presented with a formal full time software development position, it compelled him to leave a traditionally mechanical engineering specific position as a drafter and commit to the identity of of a software developer. We see how formal reification can induce a change in identity. Alex now self identifies as a software developer with knowledge of mechanical engineering.

In those days, there were just a few software programmers, most of whom were mechanical engineers. Each person would get a very large part of the whole product to make. They would occasionally communicate with each other, but mostly worked independently.

Eventually, practices like code review and common code base, revision management systems were incorporated into the community. The development team grew to about 5-7 developers. The nature of the company was starting to change. TAI was switching from an entirely government or industry research projects based company to a product and services company.

Participation: As the community of practice grows, Alex's participation in the community changes from working solo mode to a role of more interaction with the other developers.

The software development team leader, Hal, decided to move from a development team leader role to a research centric role and Philip, who was also a development team member, became the new software development team leader. Philip felt that the unofficial waterfall model of development was better suited for mechanical engineering, not for their needs of software engineering. Philip worked closely with Alex over time to convert their development practice from a loosely structured waterfall process to a structured Agile software development with Scrum and eventually also Kanban. The number of software developers grew and the structure and practices adopted within agile software development helped shape how work is done.

Participation and Negotiation of meaning: Motivated by the needs of the software development community of practice, Philip and Alex worked together to redefine what participation in the community means. Adopting a formal model of software development process and committing to it reifies the process for the community as a whole and negotiates the meaning of the community.

Alex started noticing that he was often the most experienced person on most teams and spent a considerable amount of time helping others with their work or get up to speed. He would not get as much time as before to work on his programming tasks. Over time, with help from Philip, Alex realized that he served the TAI software development community better by helping others get up to speed and by ensuring that the work is getting done in the best way possible, considering its effect on all system components and in a way that allows it to be readable and maintainable and consistent with other overarching design choices.

Alex participates in almost all major design decisions, helps assess any potential work affecting the primary code base before the company even commits to it, sometimes works directly with clients to negotiate undertakings. He helps train the newcomers while working on teams with them, and also works on development teams with other, less experienced developers.

Identity and Participation: Alex, through participation in the community, realizes that the nature of his participation was changing and not allowing him to participate in the community commensurate with his identity as a software developer. He redefines his identity, through participation and interaction with Philip, to become a mentor, a facilitator and software developer. His new identity allows him to serve and participate in the community in a way that serves the community best.

4.3.1.2 Karoline

Karoline's first degree was in mathematics and after over a decade, she went back to school to get a degree in computer engineering. She briefly worked at a big software company where the software development did not follow any formal software process. Then Karoline joined TAI as an intern software developer.

Karoline started working at TAI at the same time as a software tester, Jenna. They were both seated in the common bullpen area initially, where everyone could see their computer screens. Karoline would often feel uncomfortable sitting in the common area, where she would want to search for any of the new and unfamiliar terminology, but feel watched and judged.

The senior programmers would come and work alongside Karoline, assigning tasks, sometimes explaining code or company history and design. She worked with almost all of the senior developers, getting one on one time with them.

In three months, she was offered a full time position, given an office space sharing with other senior developers. She was also made the team leader for the maintenance team where two new development team members had just joined.

Identity and Reification: Karoline's identity changed from a mathematician to computer engineer intentionally. She then joined the software development community of practice at TAI as an intern. Her lack of privacy and her own space caused her to not feel confident in her role in the community. Her ability to participate effectively was affected. Then she received formal forms of reification like being offered a full time position and was given her own shared office, away from the common bull-pen area where interns sit. This helped solidify Karoline's identity as a developer in the community and she felt more confident and effective in her abilities since then.

Participation and Reification: After some time, Karoline became the team leader of the maintenance team, which served as another form of reification. This formal reification and change in identity also changed Karoline's participation in the then pioneer heavy community community. it solidifies her position in the community and identity as an established software developer, within a primarily pioneer heavy community.

After more than a year of working at TAI, Karoline was asked by the department manager, Phlip, to attend a high level meeting about software process, a meeting she would not typically be party to. Being asked to attend a high level meeting motivated Karoline to learn everything she could find about software process in preparation. After attending the meeting, she felt even more motivated to explore the concepts of software process. Over time, Karoline has become one of the strongest proponents of rigorous software process.

Participation: By encouraging participation in a meeting, Philip led Karoline to explore and examine different means of being in the community. Over time, Karoline's interest in software process grew to the point that she is now one of the more active and vocal proponents of rigorous adherence to software process.

4.3.1.3 Ivan

Ivan had extensive experience in the software development industry in several different types of teams and different type of software companies, from working for a global software giant, a small start-up, an open source-like software development community, etc. He also had some experience with Agile software development prior to joining TAI.

Ivan joined TAI around two years ago, shortly after Karoline, as a full time software developer. Unlike Karoline, he was given his shared separate office space when he joined.

Participation: Having worked in the software industry for a long time, he felt like his identity as a software developer was already well established. He did not require formal reification to realize his identity in the community of practice.

Like Karoline, Ivan got extensive one on one time with all the pioneer developers and subject matter experts and process experts and learned a lot about the history of the company, decisions related to overarching design, some design rationale, etc.

Participation: Ivan has worked on the UI and graphics related functionality before and wants to learn more about the physics and math involved in the application.

Negotiation of meaning: Contrary to common practice at TAI, Ivan sometimes tends to like working on multiple tasks at the same time. Ivan also tends to work on proof of concept on his machine locally before committing the changes in batches to the shared code base. This can sometimes cause some strife among Ivan and some other developers who feel that it can hinder overall team progress.

4.3.1.4 Casey

Casey is an undergraduate student, who has worked in some small software teams like student teams or worked remotely for a software team, but had not worked in full time software development before his internship. TAI is his first time working in full time software development in a co-located industry team. Casey joined the team as a software development intern.

He reports that his first few days were very stressful. He was not very happy with the 'onboarding' process and that he would have found more information about the product

and software process and code base useful.

Casey joined the Scrum development team, where he was paired with pioneer developer Quinn on a story. Quinn, who worked remotely, would spend most of the day on the phone and screensharing with Casey, walking him through the problem description and the steps for the solution. Often in the process of describing the solution and assigning it to Casey, Quinn would directly implement the solution. Casey reports that he learned a lot about the code base and design from the extensive, direct, albeit remote contact with the pioneer.

Participation and Identity: Casey found himself struggling during his first few days in the community of practice. He felt ill-equipped and watched. He did not feel like he belonged to the community. He felt that he did not know how to participate and that his identity was not related to that community.

That changed over time as Casey got extensive one on one mentoring from pioneer Quinn, who taught Casey how to participate in the community of practice. Also, by associating with Quinn who is a pioneer, and learning directly from him, Casey's identity in the community was established. Equipped with knowledge and direction from Quinn, Casey felt more comfortable and confident, and finally felt like he was a part of the community of practice.

4.3.1.5 Philip

Philip has a degree in mechanical engineering and has even taught at the university level. He started working at TAI as a software developer and worked on the team under Hal. When Hal decided to move to a different area, Philip was asked by the founders if he would be interested in leading the team and he accepted. Eventually, Philip went on to become the head of the software development department.

Some work is in maintaining and enhancing the primary code base, for their central range of thermal analysis products. There are typically two to three development teams of four to seven developers each and occasionally one team two to three developers working on maintenance on the primary range of products. There is also a team working on a new product line altogether, which is a different code base. This team is in the new product development phase. It works closely with new and potential clients to help design their product and to market it.

Philip also leads smaller duration projects and contracts to engage new clients, which are often performed by a temporarily assembled team from the existing development teams on

a need basis.

Participation, Reification and Identity: We see that Philip's identity changed from a mechanical engineer to a practicing software developer with mechanical engineering knowledge when he enters the community of practice as a developer. When he was reified for his work by being asked to become the team leader, his identity changed again to that of a team leader with both mechanical engineering and software development experience. This changed his participation in the community as well.

Over years, Philip worked with Alex and chose to switch the software development process over to Agile software development. He intentionally grew the development team in size and organized them into separate sub teams, following Scrum for main product development and Kanban for maintenance. As the skill set grew, it became necessary to have a formal system that facilitates knowledge and skill management, thereby adopting the organization of skill teams into chapters and guilds.

Philip, with input from others, determines the hierarchical skill organization and reporting structure of the software development department. Philip is also participates in making major strategic and tactical decisions about the direction of the company. Philip also meets individually with all members of the software development team to discuss their concerns and help frame a path to achieve their personal goals. Philip usually attends the daily stand up meetings of all the development teams as the client representative, and is the proxy client for many different projects or features that the team works on. Philip decides which developers should be part of which projects or should share their input in endeavors besides their primary project work.

Philip also holds occasional meetings where he introduces a new process or new tools to the developers and encourages them to use it. Philip also encourages the developers to share anything they use or find interesting and which could potentially benefit other developers or encourage conversation about their own processes, practices and tools through knowledge sharing.

Negotiation of meaning and Participation: By intentionally introducing Agile software development process to the community of practice and then years later organizing skill teams into chapters and guilds, Philip changed how participation in the community of practice took place and changed what it means to be a community for all the other community members as well, even though his own participation was different from the other developers.

4.3.2 Themes

We study themes of changes in process over time to accommodate the growing team or the changed focus.

4.3.2.1 Participation and Negotiation of Meaning: Internship changes over time

The internship program and what was expected of the software development interns drastically changed in the community of practice. In the earliest days of the company when Alex was an intern, he could work a few hours a week, alongside his other coursework. His time at the workplace was largely self-determined as his work was primarily performed alone. There was occasional interaction with the other developers, but solo work almost all the time.

After the adoption of Agile software development process, the was the developers participate and worked together changed to a model which was interaction heavy. To effectively contribute in the community, the means of *participation* now included the day starting with a daily stand up meeting and required many points of interaction throughout the day, including paired programming and design meetings, requirements meetings, customer proxy demos, etc. The earlier model of participation for a few hours a week would not serve the newly *negotiated meaning* of what it means to be a community of practice.

So the internship model was changed to have interns spend a full forty hour work week and *participate* just like the other full time developers do, by spending all day, working together with multiple points of contact and forms of interaction with other developers, testers, customer proxies, Scrum masters, Product Owners, etc.

4.3.2.2 Trajectory: Pioneer's journey

Alex was recruited by Leon to work at the company as an intern while he was getting his mechanical engineering degree. The company was very small then. Only the core founding members and a few interns worked in the company.

".. I was working from three to six (pm) one day, the next day not at all. It was a part time job at that time where you are kind of setting your own place."

Alex was given the task of making the GUI. He often worked alone for a long time. *".. I was just given some tasks, go off and figure out things and gui related tasks. I was basically told that go off and do this work. I had access to the stack of x11 reference books."*

Alex then finished his degree and internship and went to work in the mechanical engineering industry for two years. Then he decided to come back and work for the company full time. *".. There were four software developers and essentially we were given a task, which was more like a project. Back then I was given the task of starting over to implement the (base) application. It wasn't so much a task as it was a project. .. It was kinda figure out what you want to do, talk it over with the three others, and then you work on it."*

This is the mode the development team operated in for *".. a large number of years.. We would meet in all hands meet, and report that we were 90% done. 90% till you are all done, whether you were done that much or not."*

Alex would typically work alone with little input from others, and rare communication within the "team". Other programmers would also work separately. Hal was in charge of the software development team. The process *".. was kinda waterfall but still ad hoc."*

Alex recollects how collaborative design was not done with the granularity that the team now adopts. *"We did some high level design and made tasks that were bigger than the high level stories we do now"* and how the team operated. *"It was a team, but it was a loose team and you get a lot of autonomy. One of the ways you get all that is if the people working on it are the de facto experts. But it could take five to six months to release, integrating would take a lot of time."*

Once it was decided to start integrating the code all the developers had been working on separately, several integration issues would arise and new bugs would be discovered. This was also the case for running their software on different platforms. Before release time, bugs specific to platforms that the original development was not performed in would be discovered and sometimes some features would have to go back to the design phase. Most of the work was contract funded and specific to a particular funding agency's needs. So the team would sometimes be supporting features no one was using anymore.

As the code base grew in size, the team started using better tools to manage their software like version control and build systems. And incorporating practices like code walkthroughs and reviews. Hal eventually transitioned out of the software development role to a research oriented role. Philip who was also part of the team was asked to lead the team. He eventually brought in a radically different process - Agile with Scrum for the development team. Alex recollects that it took *".. it was many years worth of learning.."* to get fully engaged in Agile.

Alex found that *".. the way that we do in agile here had helped tremendously .. specially from a management side, of being able to see when you are ready or getting a true state of the application."* Overall, Alex finds that Agile was successfully adopted and served its purpose of being able to efficiently manage and produce code, however there was a tradeoff, *"We kept the ability to say how are we going to do it, not whether it is done."*

Over time, Alex becomes one of the few people - Alex, Leon and Quinn who know the code base almost completely and have been involved in most major design decisions. Leon started working on other projects and has been away from the traditional development team work in the code base for a few years.

Alex realizes that he tends to be the most experienced person on the most teams, and a lot of his time is spent helping others. He gets less time than before to do his own share of the programming work. He has become a silo of knowledge, perhaps to his own disadvantage. *"I used to complain to Philip that I don't get time to do my job. Because I defined my job as writing code. It took me three to four years. I realized that my job is not exactly writing code but in getting the team writing that code, testing and development done."* Over several discussions and reviews, Alex realized, with guidance from Philip that his role is not just to work on his own code anymore, it is to enable the team to produce good design and quality code.

As Alex gets comfortable with his newly realized role, he talks about changing his strategy for task management. *"I have evolved a pattern that if we are working on a scrum team with multiple people on the same story, unless necessary, that I will not take up a task, if I cannot do it in a timely manner, it will become a blocker or dependence for others"*

The new strategy is to mitigate the risk for contention of resources, where the resource could even be time with someone like Alex. *".. I have to have a plan in place for what we are going to do for this story, figure out what the sub tasks are and try not to take the ones that, if this one doesn't get done, then it will prevent other tasks from being done."* Following his own strategy for resource contention mitigation, sometimes Alex has to have the greater vision and intentionally choose to not do something he enjoys. He sees his responsibility *"to kind of be cognizant of what others are doing and even if it is something I would enjoy doing, back away from doing it."*

Discussion: We notice that even though Alex did not originate from a software engineering discipline, he used his mechanical engineering and programming knowledge to enter this particular software development Community of Practice. He received some guidance from the existing developers but largely had to figure things out on his own.

He often worked on very large areas of the code and functionality by himself. He gained his

expert status with a lot of work over a long time. This was a phase of participation through reification, where being offered a full time position, being trusted to work independently on high impact projects defined his role and interaction. He did work with other software developers, but in a limited and sporadic capacity.

He continued to stay constantly involved in most areas of the code and participate in most major code and functionality decisions. He actively participated and supported in the change in process and teamwork practices. Through discussions with a mentor, he recognized his role in the team was changing from being defined largely by *reification* like produced code to being defined more by *participation* like enabling the team to make decisions and plan work better by using his expertise and experience.

4.3.2.3 Early onboarders: Divergent trajectories

We now examine some trajectories of participants joining this specific software development community of practice and learning from the pioneers but forming their own distinct identities through participation and reification.

Background: Karoline and Ivan have been working in the software development team for over 2 years and joined the company at about the same time. Ivan had extensive and varied experience in the software industry and joined the team as a full time software developer. Karoline had relatively less industry experience and joined the team initially as a software development intern.

Alex reflects on how the internship model had changed due to introduction of process. *"Now how the internship works is that just dropping in and out is not effective, there is so much more communication."* Interns were now expected to work the same amount of hours as full time employees would.

In their initial days at the company, Karoline and Ivan got their information for the job from the pioneers. When they joined, there was one software development team and most of the onboarding process was to get dedicated time from the experts to learn the ropes and the history of the software.

Placement: Ivan as a full time employee was given his designated shared office space and Karoline as an intern would be placed in a common open area seating which is surrounded by the more private offices.

Karoline, who joined as an intern describes sessions of sitting in the "bullpen" like open seating area where her computer screen was fairly visible to everyone in the team. Occasionally, more senior team members would come and sit next to them and give them a task to do. Then the senior member may sit on the computer next to them for the day. *" ... Alex would come up and sit right next to you. .. it was terrifying.. I mean initially I didn't know what to do and I would google everything I didn't know and they are right next to me, and I felt like that was quite awkward"*

Karoline reports that this was both helpful and in some ways, nerve-racking as she would feel very visible to everyone, almost watched. She would be embarrassed that everyone could see her possibly fumble and attempt to figure out the jargon and the task assigned to her. However, she says that sitting and working with different members of the team was extremely helpful and she learned a lot about the code base from the different pioneers as they all offered a different perspective based on their expertise. She also felt embarrassed to ask many questions, not knowing if what she was asking was common knowledge or something not many others would know.

Ivan recollects spending long sessions with the senior programmers learning in depth about a topic about the product or code base at a time, along with the historical rationale for making decisions. He said he found those sessions very instructive and helpful.

Ivan describes what most knowledge transfer was like when he joined, *" The general feel was that Philip, at least, had more time for one on one interaction. Alex had time for one on one interaction."* and what aspects of his interaction with the pioneers helped him. *"Initially I felt that being able to ask Philip more general and esoteric questions about the application - like the history of the components, the trajectory of development in the past, I felt that was useful for my perspective stepping in. "*

Formal reification: Karoline getting her own office space, and not having to work in the common bullpen area made her feel accepted into the community. Karoline reflects *"..What's fun about it is when I got an office and I still got an office with Alex, but I was really like .. with my computer."* Karoline also shared that being made in charge of the maintenance team where she was responsible for onboarding other newcomers made her feel accepted, appreciated and more confident in her abilities. Karoline says *".. it definitely made me feel responsible. I didn't have someone else to defer to. It's scary at first for sure."*

Participation: Ivan describes how an external consultant who engaged the team in different activities, about a year after Ivan started working at the company, made him feel engaged with the team members. Ivan says *" ...there is comfortability among the workers that you get from banter and pleasantries and personal conversation."*

He adds that feeling needed professionally helped him feel like a part of the team, *" .. there is also your sense of being on the team that you get from professional interactions like going to somebody for expertise or people coming to you with questions builds that sense of like, oh, I am needed or this is who I go to or I go to this person for these questions.. I guess just knowledge of resources - human resources and brain trust resources and feeling once in a while like you can provide some value to someone else as well, really makes you feel like part of a team"*

Encouraging process proponents: Philip asked Karoline to attend a high level discussion meeting about software process, a meeting she did not think she would normally be invited to. This made her curious about software process, and she read up on it in preparation for the meeting. She found that being included in that discussion has made her an even stronger proponent of software process than before.

Process negotiation: Karoline and Ivan may sometimes disagree on process and communication expectations but through the retrospective and a third party mediator, they are able to come to a better understanding of their differences in expectations.

Ivan sometimes enjoys problem solving on different tasks at the same time. Karoline, who is a software process proponent, is often of the stand that team members should typically finish and commit their current work or current task before moving on to the next task, unless there are obstacles or external delays in the current task, allowing others in the team to take up the next task when they are free and avoid bottlenecks. In the past when working together, Karoline and Ivan have had disagreements about task management within the team. Through discussions and even third party mediation, they have come to understand their differences in expectations better.

Karoline reports that since Alex and Calvin have been working on the team with Karoline and Ivan, that Alex and Calvin frequently work together with Ivan to ensure obstacles to task completion are collaboratively handled, mitigating Karoline's stress over task management.

Mentoring styles: Some team members reflect on different mentoring/teaching styles of different team members.

Karoline says about Alex's presence in the team when she first started working *".. (Alex) is a calming influence, anytime he speaks, everybody listens .. everything goes very smoothly when he is involved."*

Karoline reflects on her experience of Alex's critiquing style. During a code or design review, Alex consciously prefers to pose potentially failing scenarios as questions for whether

the current solution would account for it, instead of saying outright what is wrong. Karoline finds that Alex tends to authoritatively question code choices which make the less experienced programmer feel like they made the wrong code choice. The less experienced programmers can feel put on the spot.

In contrast to Alex's code review style, Leon, who is just as experienced with the code base has a different reviewing style which is more casual. He tends to say, "I think this is good, but I would suggest substituting this class for that here". Karoline and Natalie appreciate all they learn from Alex's technique but prefer Leon's style as it feels more casual and conversational.

Karoline also reflected that Oswald, who is a much younger but still experienced member of the team, has a very different teaching style. When asked to explain some concept, Oswald would explain like one would to a child, breaking it down to the basics and not making assumptions about what the other person might know. Karoline appreciates this style of explaining. Calvin has a different code critique style where during code review, he would point at things and say "This is wrong" or "That is bad". Karoline appreciates that Calvin helps make her code better by pointing out flaws, but says she often has to ask many questions to get at what specifically Calvin thinks should change. The process can often be long.

Similarly Karoline finds that when asking Alex to explain something, she often has to ask many subsequent questions to get at the full explanation, as Alex tends to explain sometimes assuming that everyone else also knows the context of the solution, which is often very familiar to him. Leon however, is not very talkative on the phone and it takes several questions to get a more detailed explanation from him, but in person he is able to communicate more easily and is more forthcoming with his explanation.

Discussion: We notice how what helped Karoline assimilate and feel accepted in the community was formal forms of *reification*, like being offered a full time position and being made in charge of the maintenance team where newcomers would be onboarded or being given an office space. This may be due to the fact that Karoline entered the community with less industry experience than Ivan in a temporary internship role and was immediately surrounded by very knowledgeable and confident pioneers.

In contrast, Ivan entered the community fairly confident of his abilities from his vast industry experience prior to joining this company and entered as a more permanent full time employee. So perhaps he did not need any more formal reification to feel assimilated. He felt more a part of the team from the informal *participation* and banter in the team and from being asked questions professionally and feeling needed.

4.3.2.4 Onboarding: The next generation

Natalie, Nathan and Casey joined the company a few years after Karoline and Ivan. At this time, the software development department had undergone a significant expansion. There were now four development teams working across two primary product lines. There was now a dedicated Scrum Master and dedicated test and documentation resources.

Placement: Nathan joined as a full time employee and Natalie and Casey joined as software development interns. Nathan and Natalie joined the maintenance team that follows Kanban. This team was initially located in one room with IT and dev ops specialist Moss. Casey joined one of the development teams that follows Scrum and was located in an adjacent room. This room seated all the other developers, test and documentation team members along with the Scrum Master and Product Owner.

This batch of newcomers got their knowledge from a mix of "secondary sources" - comparatively newer people teaching newly hired people creatively like Ivan, Karoline, Davin - relatively less experienced and Quinn - far more experienced.

Reification: The newcomers are asked to commit code and file a bug report within the first few days of joining the software development team. This act of reification shows them how to participate and they realize that even as early as the onboarding process, they are participants and contributors instead of just observers. Philip also recommended that the newcomers take time everyday to go through the product tutorials to understand how the product works.

Secondary source mentoring: It was originally planned that there would be two members, Davin and Aurora when the two new members join, and Aurora would be responsible for "onboarding" the team. However, Aurora could not be there for the first day that the interns were joining and Davin was suddenly responsible for onboarding the two new employees in the maintenance team. Davin had been working at the company himself for a little over a year, including his internship experience.

As part of the role based onboarding, Davin, who was left to creatively share knowledge in the absence of Aurora, gave Natalie and Nathan the option of either looking at a new bug themselves or watch over his shoulder while he works on a bug. Natalie and Nathan opted to look over Davin's shoulder while he worked on a bug.

Natalie and Nathan worked together, either looking over Davin's shoulder while he worked, thinking aloud or working on a separate bug. After Natalie and Nathan spent some time

investigating, their bug, Davin sat with them, looking over their shoulders and guiding them as they continued their investigation. Davin then discussed options for a solution and gave Natalie and Nathan a lot of help implementing the solution. In the process of working through that first bug, Natalie and Nathan came across their first bug as well. They spoke to the tester about it and she confirmed that it was an unreported bug and gave them advice on how to file a bug report.

Typically, Natalie and Nathan would pair to work on different bugs, occasionally also pairing with Davin. From time to time, Natalie and Nathan would ask questions of other specific development team members that had more knowledge of that area of code and occasionally they would talk to customer proxies and testers where the bug behavior was not clear. In this way Natalie and Nathan had limited and sporadic exposure to more experienced developers and pioneers, but did not work with any for an extended period of time.

Ivan reflects on how the onboarding process changed from how he experienced it *"I think as the team grew, instead of spreading thin, I think that a lot of the interaction didn't occur at all and hopefully that hasn't been too detrimental... Hopefully some of the less experienced engineers, that was Philip's intention and plan, pick up that slack. The new hires and interns tend to rely more on everybody as opposed to a few. And so, we(secondary sources) are not as familiar with providing that kind of direction, nor do we necessarily know all the answers to questions that an experienced engineer would."*

Pioneer and secondary sources mentoring: Casey reports feeling very lost and stressed the first couple of days at the company. He felt overwhelmed with the jargon and trying to understand new tools and systems all at the same time. Casey joined the Scrum team and worked with senior programmer Quinn on a small story for a while when he first started on the team. Quinn would be on the phone and screen share with Casey and would walk him through the problem and solution for the new feature. He would also work on the code in front of Casey so that Casey could follow along Quinn's work. Casey found the extended one on one sessions extremely helpful and started to feel like he knew how to start working with the code base more easily.

Team reorganization: After about a month and a half from the time the newcomers joined, the teams were reorganized. The maintenance team following Kanban was renamed to be the pre-release team and many members of two of the Scrum development teams were placed in the pre-release team, to prepare for a major version release. The pre-release team became larger with about six to ten members at any given time.

As part of this team, Natalie and Nathan got the opportunity to work closely with other

developers more regularly. Here they had access to one pioneer Alex and sometimes another highly experienced member Quinn. And other secondary sources like Karoline and Ivan. Casey who also joined the pre-release team got to experience what it is like to work in vastly different areas of the code and to work on smaller pieces of work, where most bugs were smaller than user stories worked upon in the Scrum model.

After almost two months of working in that model, the teams got reorganized again. A new Scrum development team was formed, that included Alex, Karoline, Ivan, Natalie, Nathan and Casey. At a later time, Calvin also joined this team later as Nathan moved on to work on a different team. In this model, Natalie and Nathan got to work on larger stories for extended periods of time, which was a different style and pace of work than most of their Kanban work.

Natalie reports that she enjoyed the pace of the Scrum work more than the Kanban work, as she got to further her knowledge and really understand something and work more closely with more people in the Scrum format.

Participation: She felt that the larger but close knit team in the Scrum format made her feel more like a part of a team and more confident. In the Kanban format, she felt that she got to speak to many of the developers, but in very minimal capacities and did not feel like she got to properly pair and work with anyone except Nathan. So the smaller team Kanban work felt more like working independently in a team as opposed to working together in a team in Scrum.

Reification: Natalie also reports that working on stories which were new features or new enhancements to existing features felt more rewarding than the bug fixing mode, perhaps because in the Scrum team, her work felt more creative and she produced a more significant volume of code for that amount of work whereas in the bug fixing mode, she may work for a whole day to find the cause of a bug and fix it with just one line. The work felt more like critiquing and correcting than creative.

The three newcomers recognized that one of the disadvantages of not having access to the pioneers was that the overview of the product, the code architecture and organization and the historical context and rationale behind major design decisions was not obvious for a long time. Casey reflected that the lack of detailed documentation where one can go and learn some of these things initially hampered his ability to confidently understand the code and its organization.

Discussion: We find that for the newcomers to the community, the change in team size and organization meant that they did not always have as much free access to the pioneers and primary sources of knowledge. They had sporadic access and took longer to understand

things like the reason behind certain design decisions and the historical context of the code organization, etc. The early onboarders got this information from the pioneers not so long ago. Perhaps in the rapid team expansion and segregation, some of that historical context and rationale that Ivan and Karoline found so useful was not written down to be formally passed down the generations of developers.

4.3.3 Events

Here is a description of events that show how these different developers engage with each other and the challenges they face and work through together.

4.3.3.1 Process tug of war

There is occasionally some negotiation between Ivan and other team members like Karoline, Alex and Calvin. When working in the same team, Ivan had assigned himself multiple important tasks at the same time. Team members like Karoline or Alex expressed some concern initially about those tasks being such that other tasks would be dependent on them and consequently held up. After discussion with the whole team in the daily meeting, Ivan decided to take up the two potentially bottleneck causing tasks.

When all the non dependent tasks got taken up, and the two bottleneck tasks were still in progress, preventing further work from being done, Alex and Calvin spoke to Ivan and helped him determine how to share some of the work to dissolve the bottleneck. Some of it was to convince Ivan to submit the work he had already done locally but not committed to the shared code base, allowing others to continue with some of the other tasks.

In the past, Karoline and Ivan had clashed over a similar situation, where Karoline supported strict adherence to process to prevent one person working on mutliple tasks as the same time and Ivan wanted to take on multiple tasks. In that instance, the negotiation to distribute work did not occur during the sprint. At a later review of the sprint during the sprint retrospective, there was a disagreement about process, which was then resolved with third party mediation.

So later when Alex and Calvin resolved the task allocation issue with Ivan, it was a relief for Karoline to not have to attempt to manage that communication as before and see the team actively manage task allocation and balance during the sprint, and working together

to resolve issues early on.

Through these events, we see how the unwritten process evolves through experience and communication.

4.3.3.2 Team deciding work for itself

In the sprint planning team, when five new members were being added to the existing three member team and the process was being switched from Scrum, which the new five members had been using to Kanban which the existing three members were using. A meeting was organized to discuss protocol and agree upon practices where developers shared their concerns and brainstormed solutions for effective team communication.

The Scrum Master and the software department manager shared their concerns with the team and advised them to keep a small work in progress limit to allow for true paired programming and collaborative opportunities.

The team however felt that a small work in progress limit would leave them bored and would leave gaps in the flow of work. So in spite of the advice they received, the team reminded the others that it is supposed to be a self managing team, and that it chooses to have a larger work in progress limit. This was an instance of the team asserting their vision for how they choose to operate, even though it is not the same as the manager's vision.

4.3.3.3 Learning to work with different mentoring styles

Alex and Leon with very similar levels of experience have different mentoring styles. Alex and Leon have worked together in the company since it was established. They both have intimate knowledge of the code base and for well over a decade, they were both actively involved in almost anything to do with the primary shared code base. About two years ago, Leon started working on enhancing certain specialty features of the code through research and was not working in the main developers team anymore. Leon's specialized research meant he was working with other collaborators and not the primary product's regular development. Alex worked with the product feature development throughout.

The comparatively less experienced developers noticed that when working with Alex, they feel sometimes put on the spot to explain their rationale behind certain decisions or to

answer what their code would do under certain situations that the developer may not have considered in the original design. All the developers know that Alex does this to make their code more robust and consistent with the rest of the code base, but they sometimes still feel put on the spot. Alex says he intentionally asks questions about what the developer's code would do in different scenarios as he doesn't want to come right out and say "This is wrong, change this." Alex uses the question asking method to spare the feelings of the developer and to allow them to come up with the change themselves.

However, with Leon, the same type of conversation tends to be more casual, where if he wants the new developer to do something differently, he will say he likes their code and will offer a straightforward suggestion asking for something to be changed. Surprisingly, many of the less experienced developers prefer this approach, as they feel less scrutinized.

Some of this could also be a result of how Alex is almost always colocated with the developers and Leon works remotely and primarily communicates over the phone, so he may feel like he does not know the developers as well as Alex.

4.3.3.4 Knowledge transfer

Among the primary development team members, Alex and Quinn have the most knowledge of the code base. There are other developers who also have as much knowledge but have moved on to other projects to use their expertise. The company has expanded in the last few years and has acquired many different customers and different types of contract work.

One of the many motivators for hiring new developers and expanding the team was to facilitate knowledge transfer from Alex to other members. Often, when Alex is working on a task with another developer, the other less experienced developer participates but is usually aware that there is a lot to be learned from that experience. There can be times when the less experienced developer may watch as Alex is coding. Typically, when Alex is part of a team, he usually does the design for most stories and shares it with the rest of the team in the design phase. With his vast experience, Alex is best suited to take the lead on the design, as he would be conscious of how the design affects other components of the software and he would also be able to implement the design fastest, as he has knowledge of the entire system and would not have to spend time looking for the most appropriate place to implement the solution.

However, this also means that the relatively less experienced software developers are not able to get as much experience coming up with design. The process of distributing knowledge and moving away from the silo model of knowledge is slowed down.

4.3.3.5 Minutiae disconnect

Alex will sometimes ask less experienced developers to redo a piece of working code to be consistent with how things have been done in other parts of the code or due to other overarching coding decisions.

The less experienced developers are often told to redo their code at the time of code or merge review. This often makes the other developers feel less confident about their choices or that whatever they decide now will not matter as much, as they will be asked to change it later.

Alex's suggestions are always followed, and are often very helpful. However, when other developers are sharing their code with Alex for review, it is often at a stage that the code does what it is supposed to functionally. Often the changes that Alex is suggesting are not in functionality, but in a style or in using a different but equivalent component to perform the same task. This type of change often requires extensive rework, to accomplish the same task.

Sometimes some less experienced developers feel that their code and design choices were valid and that the suggested changes are unnecessary. That their choices are sometimes trumped by Alex's stylistic choices, even if they are just as good.

Overall, the lesser experienced developers feel like they do not have as much ownership of the code base as Alex does, as his choices often override theirs.

4.3.3.6 The burden of onboarding

All the developers in the software development department have linux workstations. The department manager and others saw value in having at least one developer on the windows platform. Most of the clients use windows and almost all the development is done on linux. This leaves primarily the automated testing in Windows to be the only windows use and exposure the product gets in house. Having a windows development machine means that more scenarios than just those in the automated tests are exposed on windows machines. It also means less overhead when there is a windows specific feature or bug being implemented, as most developers have to perform some set up to run, develop and test on Windows.

Karoline was put in charge of the maintenance team when it had some new interns. This was a new experience for Karoline, to lead a team. Aurora, one of the new interns in the maintenance team then had a microsoft workstation, as was decided by the upper management. However, an intern learning the ropes on windows machines posed other challenges for both the intern and Karoline, as many standard tools and environments would not be as easily set up on windows. Most of the in house instructions for setting up and accessing different systems and tools were also only written for the linux workstations.

It often meant that even Karoline or other developers did not have helpful answers for Aurora when she encountered problems setting up her system. It also meant that the application sometimes behaved differently on Aurora's workstation than the rest of the team's. It made it harder for her to feel like she is up to speed as easily as others and it made Karoline feel less able to help her team member.

4.3.3.7 'Pair' programming over time

It is very common and highly encouraged as part of the company culture for developers to 'pair' on a task. Until the year before our observation window, this form of 'pairing' often involved the developers discussing the task or design, then dividing up the sub-tasks between them, and they go and work separately and eventually regroup, to discuss further and divide tasks further.

Over the last year, the software development culture is starting to change. When the three newbies joined, two interns and one full time employee, some of what was considered 'pairing' started to be redefined. Davin who was onboarding two of the newbies asked them to look over his shoulder as he works and then he looked over the newbies' shoulders when they tried to work on their first task. While looking over their shoulders, Davin guided them through the problem discovery process, and helped identify a solution while the newbies drove the screen navigation.

This different style of 'pairing' continued in the maintenance team, where two people would work on the same task together at the same workstation, while each occasionally drives. Then as the newbies started working with other developers, they followed this new model of 'pairing' with them as well. Over a few months, some of the existing, more experienced developers also started 'pairing' in this new way, where they would work on the same workstation together.

4.3.3.8 Pace discrepancy

Right after Nathan's stint in the maintenance team, when he first started working in the pre-release team with Karoline, she found that he took longer than she was used to to go through the investigation step for any bug. She found that perhaps because of his lack of confidence in knowledge of the code base, he would revisit different parts of the concerned code repeatedly, but would not confidently draw conclusions for a day or two. Karoline, who is far more conversant with the code base and the programming language would be accustomed to typically half a day of investigation at the most. In working with Nathan, she would identify the cause of the defect quickly and recommend a solution for it. However, Nathan would continue investigating, unsure of the root cause or solution. He would not be confident that a subroutine being used is actually doing what it should and start investigating subroutine after subroutine, even ones used extensively by many different parts of the code. Then after two days, Nathan would propose the same solution that Karoline would have proposed before. After working in this mode for a few days, Karoline carefully spoke to Nathan, and asked him to trust the code he sees and not try to question everything, especially if it has been proven to work in other parts of the code.

Eventually, Nathan asked Philip for time to learn and become more confident in his language understanding abilities. Philip encouraged Nathan to take a course for that purpose. Some time after the course, Nathan appeared more confident in his conclusions about problems in the code.

4.4 Quantitative Data Analysis

4.4.1 Collecting the data

The standard company process at TAI is to record what contract related work one is working on for every part of the day, to the granularity of 6 minute intervals. Different developers use different methods to track their time usage. Some make notes on paper, some document it in a saved text file, so that their activity is searchable later. Our participant observer, Natalie initially used notes on paper and then transitioned to documenting time used as events on an online calendar. She used the hand written information from the first few weeks to backfill the events missed on the calendar.

The events were usually named to indicate whether Natalie worked alone or pair programmed with another developer. Sometimes the event names included which bug or story she worked on. When the bug or story information was missing, Natalie would use commit logs and bug reports to determine for each activity which bug or story it was related to. Events like the daily stand up meeting or a sprint demo meeting, which were related to multiple bugs or stories at the same time, were categorized as multiple.

To track which events were related to each different bug or story work, Natalie would either record part of the name in the event, or would check which story or bug was worked on that day by looking at a combination of commit logs or bug report updates or emails, to ensure only the correct story or bug gets associated with each event.

This data was collected to understand what happens in the course of a bug or story for a developer in different models of software development and through the onboarding experiences. How often do developers work by themselves, how often do they pair program, how does the software development process affect communication type and frequency?

From the data, we could potenatially answer questions like - how often did Natalie work with someone or work alone? what was the nature and frequency of Natalie's work like in different teams and different software development processes? What was the nature of Natalie's interaction in the onboarding stage and post onboarding? Keeping the nature of the questions in mind, we examined every event and described it using different aspects of the data events like the number of attendees, the type of interaction, the type of meeting, etc.

4.4.2 Description of the raw data

The daily communication data that was collected by our participant observer was in the form of calendar events titled with a brief description of what the participant did during that time. Most of the events were only for the benefit of tracking activities, there were some events that were created to invite attendees to the events, like the development team being invited to the daily stand up.

The calendar data had the name of events, like 'paired with Alex' indicating that the participant observer was pair programming with another developer Alex during that time or 'alone on bug' which would indicate that the participant observer programmed alone, working on a bug. The date and start and end time of the event, the name of the organizer, which in most cases was the participant observer, as most of the events were created by Natalie after they occurred.

Some of the calendar events were created towards the end of the day, on the day of the event. And some events were created in the calendar a few days after the event occurred. The other events were invitations from other members of the software development community for meetings.

There were some events like the daily stand up meeting for different teams or different requirements collection meetings or sprint planning meeting or demo and demo planning meetings which were organized by someone else in the team, usually the scrum master, and had multiple attendees.

4.4.3 Processing the data

The calendar data was extracted to a list, and patched with missing pieces, where some recurring events like the daily stand up had not been replicated based on their recurrence. Some events were initially named inconsistently and were renamed to match the style of other event names. Most event durations are rough estimates as different events related to the same work sometimes had difficult to distinguish or overlapping time boundaries.

4.4.4 Processing the data by hand

Then for each of the almost 550 events captured, the participant observer went through each event to categorize it in terms of – Meeting Type - whether it was a 'Pairing' session where the participant observer was pair programming with another developer, or a 'Solo' programming session or a 'Meeting' of multiple team members, or a 'Presentation' like an information session or a demo, where one party presents information to the attendees.

Tables 4.2 and 4.3 on pages 84 and 85 describe the different attributes and categories used to categorize each calendar event.

For 'story type', the attribute category Company is used for events that are related to general company work or welfare like a general meeting; User Stories is used for events related to user story work in a scrum team; Onboarding is for events related to the initial onboarding of the newcomer; Bug is for events related to work on a bug; Kanban is for events related to kanban process; Planning is for events related to planning; and Team is for team related events.

The attribute of ‘attendees’ represents the total number of attendees for the event in question. In some cases this is an estimate, like for the general meeting where all company employees attend, we estimate a number like 30 whereas the number could be closer to 50. These events would typically not be linked to any bug or story. Other meetings would typically only have the a subset of the developers on the team

‘Story ID’ is a unique identifier given to each story or bug that potentially multiple events could be associated with. This identifier was only used to compare events and other attributes of events across different stories.

The attribute ‘CommType’ describes what type of communication the event is associated with like whether it is a meeting where different people participate or a directed presentation or solo work, akin to the Alone Meeting Type, of if it is a pairing session or a consult with another developer, or some combination of solo work with a brief consult.

The ‘Directed?’ attribute reflects whether the style of the communication in the event was directed, like in a presentation or undirected, where different parties communicate with each other.

The ‘Structured?’ attribute describes whether the communication in the event followed a pre-determined structure, like a daily stand up meeting has standards about procedure where all the development team members answer the three questions each. Unstructured meetings tend to be freestyle. Similarly the ‘Formal?’ attribute describes whether the style of the meeting was formal or not. Sprint demos tend to be formal and structured meetings, whereas paired programming structures tend to be informal and can be either structured or unstructured.

The ‘Repeated?’ attribute describes whether that event is a one off or a regularly or irregularly repeated event. The ‘Ritual?’ attribute describes whether the event is part of a ritual like something that happens at the beginning of the sprint or the end of the sprint or a before story start ritual or a reflecting ritual.

The ‘Expert present?’ attribute describes whether in the interaction in the event, there is an expert or authority on the current knowledge discussion present. Like a pioneer developer present at a consult session or a customer proxy or requirements expert present in a requirements discussion.

The ‘Information Flow’ attribute describes the flow of information and the kind of information flowing in the event, like if the information is discovered together by the participants in the event or reported by a party to others like in a daily stand up meeting where every development team member reports on their status; or whether information is exchanged or

Table 4.2
Different categories and attributes for processing the event data

Attribute Name	Description – Attribute Categories
CommType	The type of communication interaction of the event. – Meeting, Presentation, Solo, Pairing, Consult+Solo, Party, Consult, ClientDiscussion
Story ID	An identifier that connects each event to a story or bug, if different events are related to the same story or bug. 0 otherwise. – A specific story number or bug number, 0 - if not related to any bug or story work, Multiple - if related to multiple bugs or stories.
Story Type	Type of story the event is associated with. – Company, User Stories, Onboarding, Bug, Kanban, Planning, Team
Meeting Type	Format of the meeting – Alone, General Meeting, Demo Meeting, Meeting, Setup - related to onboarding, Programming, Using - learning to use the application, Design, Discussing, Estimating, Planning, Reflecting, Requirements
Linked to story?	Whether that event is linked to a story or bug, or unlinked.
Attendees	Total number of attendees for the meeting
Directed?	Whether the meeting is a directed presentation, where one party is directing information towards the attendees.
Structured?	Whether the format of the meeting was structured. Whether it followed a predetermined set of rules.
Formal?	Whether the format of the meeting was formal or informal.
Face2Face?	Whether the meeting is face to face or telephonic or a mix of both.

Table 4.3
Different categories and attributes for processing the event data

Attribute Name	Description – Attribute Categories
Repeated?	Whether this was part of a repeated event – Y- repeated, One-off, Y-Irregular, Impromptu, Monthly
Audience?	The audience of the event – All, Design+Dev, Newbies, Developers, Team, Self, Client+Developers, Modeler+Developer
Ritual?	Whether the event is part of a ritual – Monthly, End of Sprint, Onboarding, Pairing, Daily, Demo, Reflecting, BeginningOfTeaming, StoryBeginning, BeforeStory, Coordinating, StoryEnd, Assigning, Pre-Sprint, SprintStart
Expert present?	Whether an expert was present in the meeting. The expert position is subjective, depending on what is being discussed.
Information Flow	What type of information flow occurs in the event – Reporting, Informing, Socializing, Instructing, Discover together, Exchange, Self, Reflecting, Clarifying, Ratify, Estimating, Planning, Devise together
Team	Which team I was a part of when that event occurred – Maintenance, Pre-release, Sprint Cycle 1, Sprint Cycle 2, Sprint Cycle 3
My Participation	Whether my participation was active or passive.
Activity Type	Type of activity that takes place in the event, this is a mix of CommType and some specific meeting names – Demo Meeting, Meeting, Setup, Programming, StandUp, Using
Duration	Duration of the meeting in minutes
Person	Names of three attendees, apart from Natalie, or Team if the team attended.

clarified or ratified like a customer demo to ratify the solution.

4.4.5 Processing the data with tools

Software process mining tools like Disco was used to reveal the structure of communication types over the course of a typical day as part of each type of team and over the course of a bug or a story for different types of teams. Disco allows the average communication flow structures to be filtered by the percentage of activities or paths identified to adjust the readability of the graphs. As the flow diagrams generated are an average of the activities used to generate them, filtering effectively creates a flow diagram which is still an average representation of the communication, but is generated from a smaller set of entries.

The social interaction graphs use different types of graph layout algorithms like Harel Korel or Fruchterman-Reingold to describe the frequency of Natalie's interaction with other developers as part of her work in different teams. Figures 4.10 on page 101 till figure 4.11 on page 102 describe the frequency of Natalie's interaction with other developers, where all developers have been placed in different groups, indicated by boxes. The group boxes indicate the level of expertise or the level of centrality to the community of practice. Pioneers like Alex, Quinn, Philip, Leon and Douglas form a group. Early onboarders like Karoline, Calvin, Oswald, Ivan and Davin form a group, which has relatively less experience than the pioneers, but were onboarded by the pioneers and have an identity that is less central to the community of practice than the pioneers but is more central than the newcomers. Natalie, Nathan, Casey, Jenna and Samuel are newcomers to the community of practice and form their own group. This group has the least experience and has an identity that feels the least central to this software development community.

Timeline creation tools were used to graph the course of events over the normal functioning of a team. Pie charts were used to graph the percentage split in duration of time spent on different types of activities during different phases.

4.4.6 Observations from the data

The pie charts depict the percentage breakdown of time spent on different types of activities by the participant observer during her work in different teams.

Timelines: The timeline diagrams show in chronological order, the activities the participant

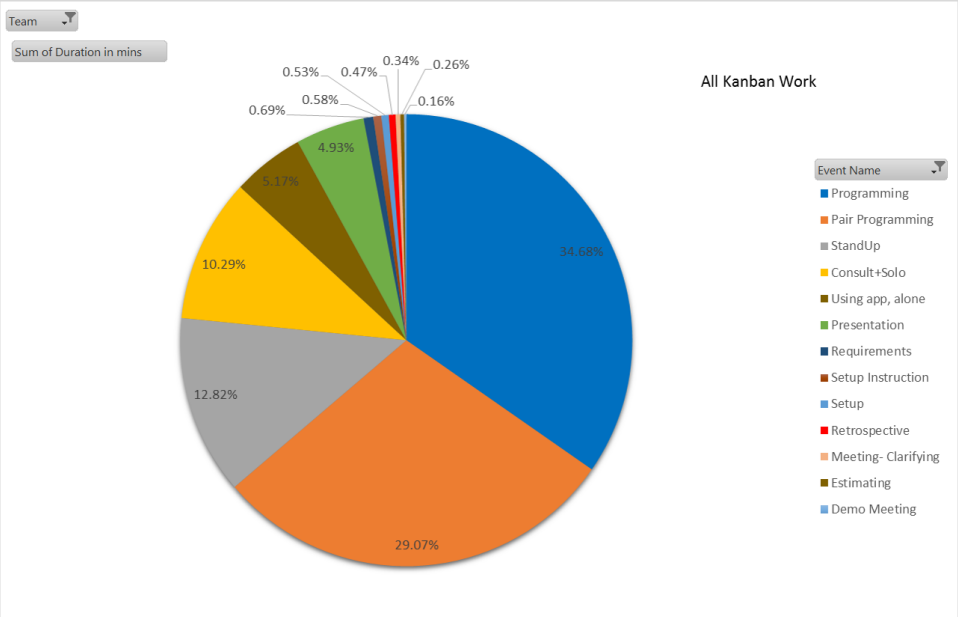

Figure 4.1: All Kanban work time breakdown pie chart

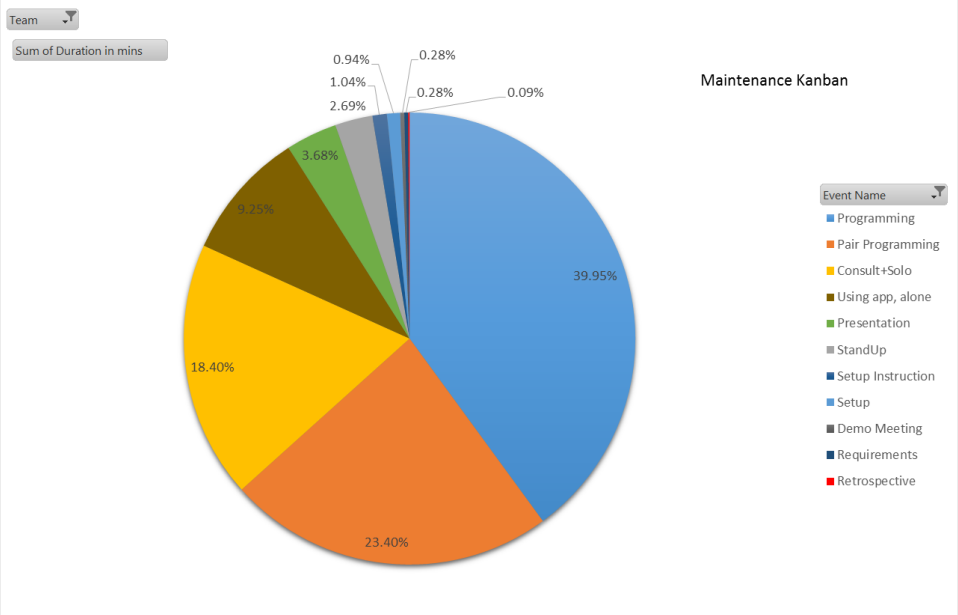

Figure 4.2: All Maintenance work time breakdown pie chart

observer engaged in during the three week period of the first sprint following Scrum 4.8 on page 99 and the first three weeks of the prerelease team work following kanban 4.9 on page 100. The bars at the bottom show the duration of different stories or bugs.

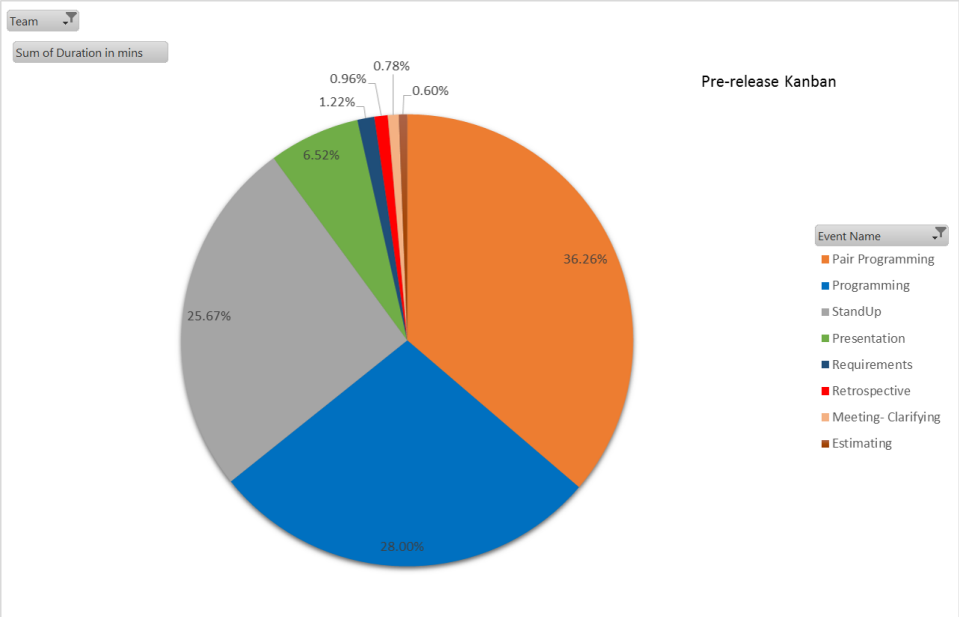

Figure 4.3: Pre-release work time breakdown pie chart

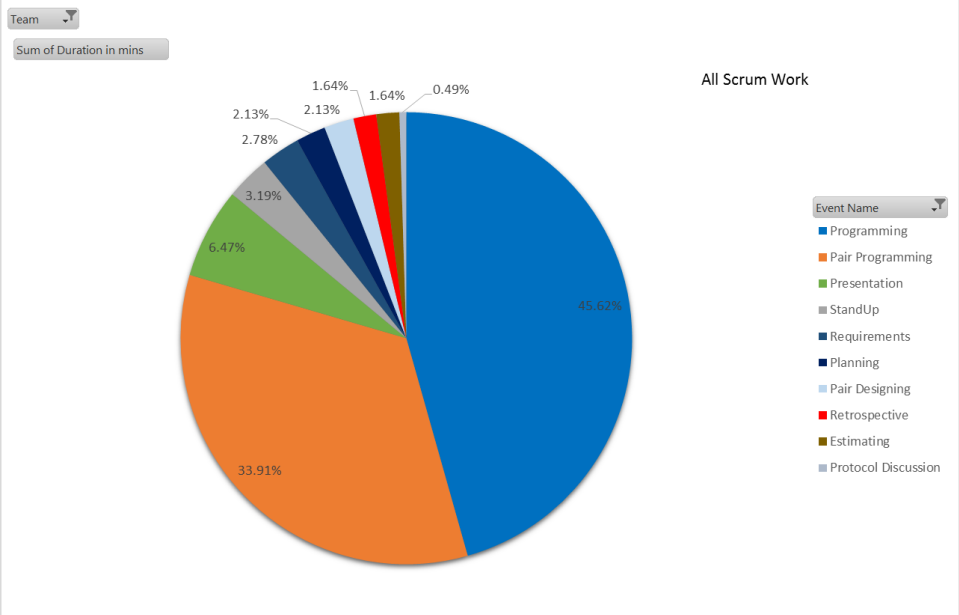

Figure 4.4: All Scrum work time breakdown pie chart

We notice that in the timeline for the first sprint, our participant observer only worked on two user stories, which took several days each. Whereas, the participant observer worked on a greater number of bugs during the first three week period in the pre-release kanban team.

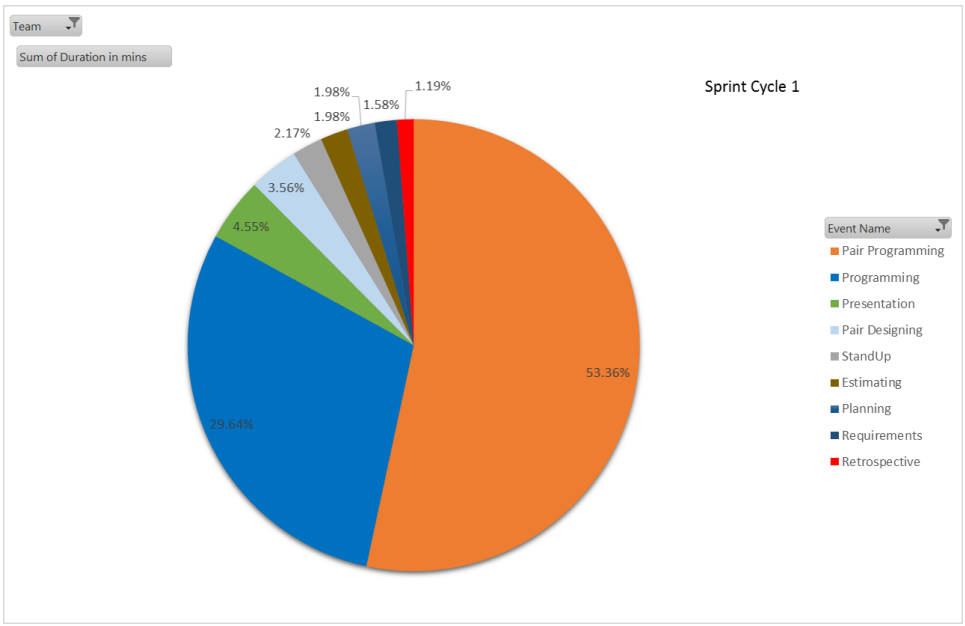

Figure 4.5: Sprint 1 work time breakdown pie chart

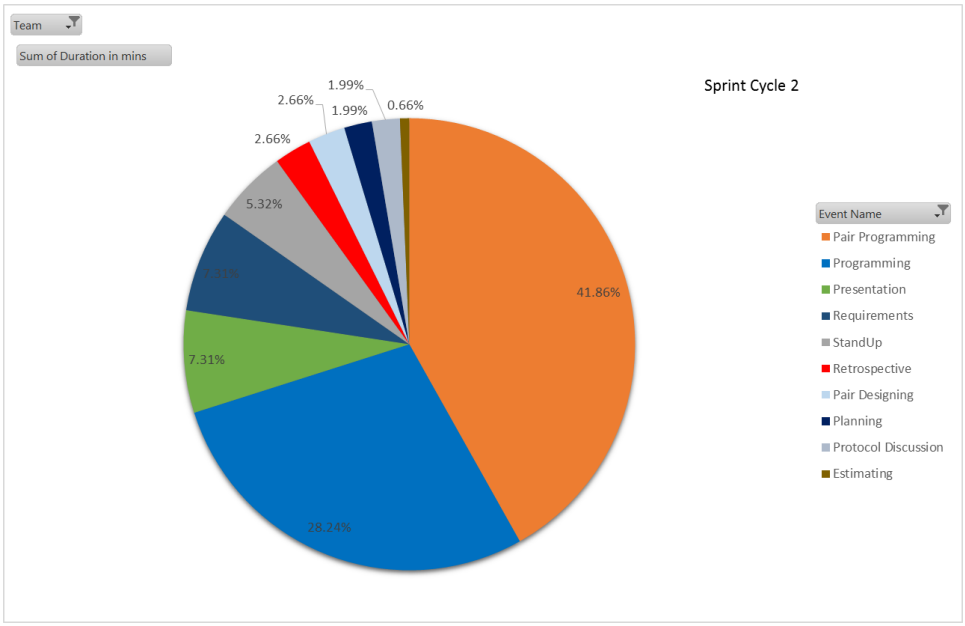

Figure 4.6: Sprint 2 work time breakdown pie chart

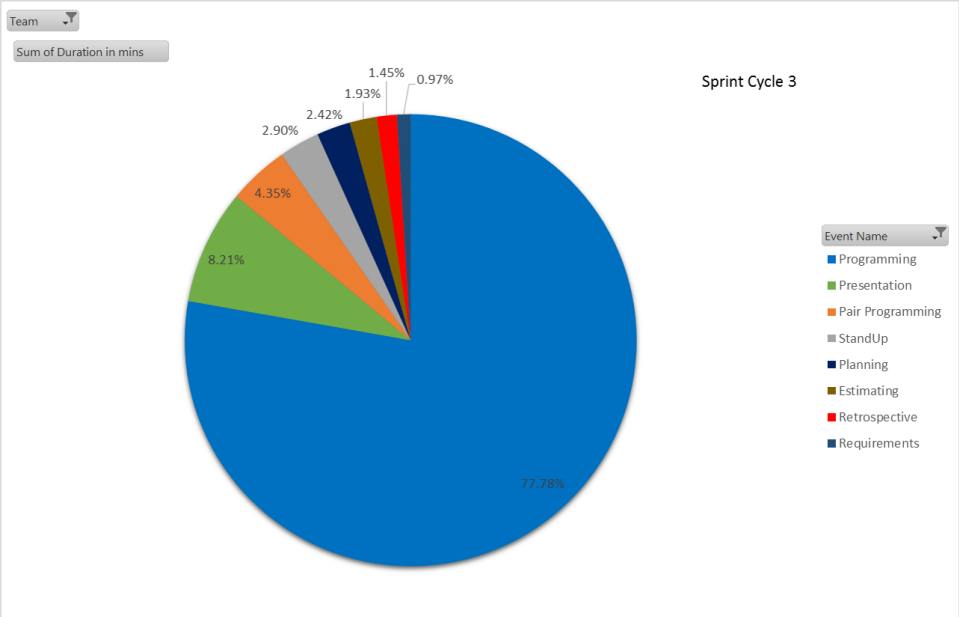

Figure 4.7: Sprint 3 work time breakdown pie chart

4.4.6.1 Story scope creep

The second user story from the first sprint depicted in figure 4.8 on page 99 almost reaches completion in the first sprint but does not make the deadline on the last day of the sprint, and gets pushed into the next sprint. The story work received a thumbs up from the customer during the first sprint, but after the demo time, therefore was moved to the second sprint to complete the final commit part of the story. However, the customer proxy for the story in the first sprint was not available in the second sprint and a different customer proxy was used. One of the developers on the story was also not present during the beginning of the second sprint. When the story was presented to the new customer proxy in its ready to commit, reconfirming approval stage, the new customer proxy added on additional scenarios for input to the code, that the customer proxy in the first sprint had not added to the acceptance tests. This caused the scope of the story to increase in the second sprint. While working on it, a developer was assigned to the story who had not worked on that story in the previous sprint. So some time was spent bringing the newly added developer up to speed for a micro onboarding at the story level.

4.4.6.2 Re-estimating

We notice from the timeline diagram figure 4.8 on page 99, that the relatively smaller story, initially estimated to require an 'effort' of five units or be 'a five', from the first sprint lasted about a third of the three week sprint, and was not considered finished even then and took some time in the subsequent sprint. In comparison, we see from timeline figure 4.9 on page 100 the bugs Natalie worked on during the three week period of the pre-release kanban were supposed to last a shorter duration, as most bugs are typically automatically assumed to be rated as an effort of one unit. We observe that some of the bugs, however, took as long as a shorter story from sprint one which was estimated for an effort of five units. Most of the developers agreed that the treatment of bugs that were too big to be bugs in kanban and perhaps would have been more appropriately presented as user stories.

4.4.6.3 Proportion of communication

In the event data, most of the activities except the ones labeled 'Programming,' which indicates time spent programming alone, are activities that require communication and interaction with others in different forms and styles. The pie charts depicting time breakdown of activities like figure 4.1 on page 87 where more than 60% of the time was spent on activities involving communicating with others and figure 4.4 on page 88 where more than 50% of the time was spent in communication related activities depict how large a proportion of a developer's time is spent in communicative activities. All activities other than the ones labeled 'Programming' were communication related and communication intensive activities, including but not limited to 'Pair Programming', which is most often the second most commonly occurring activity by time spent.

The social interaction charts depicting whom Natalie interacted with and process charts depicting the average flow of activities that take place in each team phase, collectively demonstrate that communication based activities and interaction characterize a lot of the work that happens in all forms of software development - maintenance(figure 4.2 on page 87 and figure 4.3 on page 88) or feature development(figures 4.5, 4.6 and 4.7 on pages 89, 89 and 90 respectively), Scrum or Kanban, small or medium sized teams, bug squashing or feature development, newcomer onboarding or business as usual, etc.

4.4.6.4 Perfunctory and evolutionary process

When Natalie was part of the Maintenance team following Kanban, it consisted of only three developers, of whom two were newcomers. Initially, most of the work was done together, and the newcomers frequently consulted with the third, more experienced developer on most bugs. At least two of the developers would work together all day and the third would be in discussion with them on multiple occasions. The three developers sat close enough to talk to each frequently.

We see in the interaction diagram in figure 4.12 on page 103 that Natalie worked very closely with Nathan who was the other newcomer, and Davin, who was the early onboarder developer in the Maintenance team and was helping onboard both Natalie and Nathan. The connections between Natalie and Philip depict the daily stand up meeting that they were part of where Philip was the Scrum Master. The other connections indicate communication where Natalie and Nathan consulted with other developers on different bugs. As we see from figures 4.17 and 4.19 on pages 108 and 110 respectively, the typical work for a bug or the typical day of work in the maintenance team included programming alone, pair programming, occasionally a consult with other more experienced developers, attending stand up and presentations for some of the bugs.

In a team so small that worked together all the time, the daily stand up meeting was a practice that the developers felt was more for the benefit of their Scrum Master, Philip, who was also the software department head. In the daily stand up meeting, Philip would sometimes even advise the newcomer developers on which other developers they could consult with.

Daily Stand up meeting: When the team changed from the Maintenance team to the Pre-release team and three to five other developers were added to the team, the Daily stand up meeting became very useful. It was a great way for the developers to update each other on progress and plan, to decide who will work together, who wants help and what needs to be done. In the larger Pre-release team, the daily stand up served its original intended and prescribed purpose. Even though, Kanban with daily stand up meetings were followed in both teams, the change in team size and dynamics made the same practices more meaningful.

WIP limit: In the Maintenance team, as two of the three developers were newcomers, the throughput of the bugs resolved in the team was lower than it would be for three fully operational developers. In this slow progress mode, the 'Work In Progress' or WIP limit, which is a core aspect of the Kanban process, is not something the team ever had to consider or change. It did not affect the team's ability to work in any way, as the two newcomers

spent their time learning the system.

When the Pre-release team was formed from the Maintenance team, with a larger number of developers coming in, and the two newcomer developers from the Maintenance team, able to contribute fully, the need for considering terms like WIP limit to determine flow and manner of work completion arose. The team discussed at length about different options for the WIP limit and decided on one where they felt they would be able to work at a brisk pace, with the understanding that they would revisit it when necessary. The team discussed and set their WIP limit at least once a week to keep up with changes in the number of developers and the type of bugs they were working on.

Customer demo: In the Maintenance team, the developers would conduct an open demo for bugs they worked on once every couple of weeks. However, the sentiment in the team was that the demos were not very useful, as presenting fixed bugs did not seem as interesting as presenting newly built features to customers. Often in the Maintenance demo, the developers would be demonstrating normal and common uses of the product, something customers are typically familiar with. As the bugs demos were usually demonstrating the bad behavior followed by the fixed, normal behavior; the demos felt lackluster when compared to sprint demos where interesting, new features were demonstrated. The demos served as good practice tools for the newcomers in presenting their work, but did not seem very useful as demos to the customers or other developers.

When the Pre-release team was formed, the work still came into the team in the form of bugs, but the developers felt that the work seemed like medium sized user stories disguised as bugs. The team decided that it was important to conduct customer demos before closing a bug and considering it resolved, especially considering that the bugs were more involved and elaborate than the normal crop of maintenance bugs. This led to a process where the developers would capture and document requirements with the customer or customer proxy for each bug and would confirm resolution of the the bug with the customer through a demo, and would eventually also demonstrate the resolution to the larger developer and mechanical engineering community at TAI during the common demos. In their more accelerated and involved mode of work, where bugs were elaborate and often resolved core issues before the major product release, the customer demos seemed more useful and meaningful. We also observe from figure 4.13 on page 104, that in the pre-release team that Natalie's interaction changed from how it was in the Maintenance team. She did not work with Nathan anymore, she worked with a lot of different developers, and this working together was no longer just in small consult sessions, it involved more extensive pair programming sessions. Natalie now worked with Karoline, Alex, Moss and Douglass. Philip's interaction with the team changed to an intermittent supervisory role and Natalie and the team worked more closely with Steve, the Product Owner.

When the team started working on user stories in Scrum style sprints, the involved process of customer requirements capture, deliberate design and implementation and multiple customer demos, was the norm. The customer demos were more elaborate and substantial as they were often demonstrating new functionality which elicited a lot of questions, discussions and revisions. In this format of development, customer demos were a crucial and essential part of the development process and felt appropriate and necessary to all developers, and was no longer considered a perfunctory practice. As we see from figures 4.18 and 4.20 on pages 109 and 111, the per bug process in the Pre-release team has evolved to include gathering requirements and customer demos or presentations, along with the programming activities. In addition, the team attends the Stand up every day - a practice that suits their evolving team needs now.

4.4.6.5 Shapes of sprints

There are some differences in the separate phases of software development team work that Natalie experienced, in some ways defining a characteristic for each phase. We consider these characteristics a 'shape' of the software phase. We find that not all sprints are created equal.

First sprint -The initial turbulence sprint: This was the first sprint experience of that particular mix of developers, it was a departure from the Kanban style Pre-release work that preceded it. In the first sprint, the developers started new stories, some of which were the beginning of some long running, multi-part feature sets. This first sprint was characterized by lots of communication - several bursts of discussion, design, redesign, repeated understanding and questioning requirements, etc. All the developers were trying to get on the same page about process and operational details. The sprint involved lots of planning, a lot of inevitable and unavoidable planning for future sprints, even though that is explicitly not the intent. The developers try and choose different pairings and combinations. In the end, a lot less work was accomplished in the first sprint than the team planned for in the sprint planning meeting. The user stories the developers worked on, which seemed straightforward in their estimation and planning phases, ended up being more complicated in its implementation. We see in figure 4.14 on page 105 which describes Natalie's interactions during the first sprint, Natalie pair programmed extensively and mainly with Alex and Karoline on two long running user stories, as we see from timeline figure 4.8 on page 99. Natalie consulted breifly with some other developers.

Second sprint - Path to normalcy sprint: After a shaky first sprint where the developers were learning to work together in this new format and realizing that they initially underestimated this brand new work, in the second sprint, the team learns to complete work carried

over from the first sprint and finish the testing of things that they barely got the time to build in the previous sprint. In this sprint, developers starting some other new feature sets and were understanding how to improve their work strategy. The second sprint had a comparatively less need for discussion than the first sprint. It still involved a lot of working together with other developers and with more testers as well to complete work that was started in the first sprint. We see in figure 4.15 on page 106 Natalie worked with Karoline and Calvin, with some interaction with Jenna, the tester.

Third sprint - BAU sprint: In the third sprint, the development team operated like a well-oiled machine. The developers spent time wrapping up or continuing the feature sets started in sprints one and two, so a lot of the major design discussions and decisions had already been made and everyone knew what needed to be done. By the third sprint, the team knows how to work together. Everyone knows what needs to be done in the sprint, as prior sprints helped clear most of the doubts and questions. The third sprint had a limited need for pairing, the developers in the team implicitly chose to work independently more. The atmosphere is a quiet and calm, almost sleepy. We see from figure 4.16 on page 107 that Natalie had very limited interaction with other developers, as she mostly worked alone, as depicted in figure 4.7 on page 90 which shows primarily programming alone.

4.4.6.6 Process in practice - Kanban vs. Scrum

Our participant observer and some of the developer subjects worked in both the Kanban style Pre-release team working on bugs and the Scrum style development team working on user stories. Through developers reflections and our participant's observations, we found that during the pre-release work, some of the bugs that they worked on felt like they entailed work more substantial than bugs typically do, and felt more like "stories disguised as bugs". Some developers expressed their dislike for it, as the bugs being too substantial hampered their ability to leverage the spirit of Kanban. Notions like the Work in Progress limit and the In Progress queue were not functioning as thy normally would. One to two developers would be tied to one bug for a relatively longer period of time than other bugs and the queue would have one to two slots blocked for a substantial period of time. The typical pace of Kanban work is fast and the longer running bugs made some of the work feel slow.

Another reason some developers said they preferred the Scrum work to the Kanban work is that there is a clear time line and end of sprint in Scrum. The developers worked in the Kanban team before a major product release and was supposed to be in bug squashing mode with Kanban, taking work input from a large bucket of bugs. The team was told their work will end when the product owner agreed that the code quality was good enough and enough bugs had been fixed. The developers felt that they did not have a good idea of

whether they were spending an appropriate amount of time on the bigger bugs. The same work in a Scrum user story would have an estimate associated with it, and the developers would have an idea of how long it should reasonably take. The developers also expressed guilt, saying they felt responsible for delaying a major product version release. The product owner and scrum master assured to them that the developers are not to blame for the delay and they are, in fact, helping deliver good product quality, which is needed for a major product version release. Overall, the effective open-endedness of the Kanban bugs proved to be a source of stress and strife where the long-running bugs were concerned.

4.4.6.7 Multi-sprint stories

As we see in figure 4.8 on page 99, Natalie worked on a couple of stories in the first sprint. The dist story took up most of the sprint, and in the wind down phase of the dist story, work on the dialog story started. Karoline and Natalie worked on the dialog story where Alex consulted. The story was almost completed by the end of the sprint. It was developed according to the requirements and scenarios the customer proxy laid out, and demonstrated to the customer proxy where the customer gave his approval. However, the customer demo was not done before the sprint demo, so the story was considered incomplete and carried over to the next sprint.

In the second sprint, for the dialog story, the customer proxy changed as the original customer proxy was on leave. Karoline was also on leave, so Natalie was assigned the story along with Calvin, who was new to the story. When Natalie repeated the demo for the story to the new customer proxy, he pointed out additional scenarios that should be included for the story to be complete. These are scenarios that the first customer proxy had not identified. This caused an initially small story, which was considered complete to be carried over and then expanded in scope. As Calvin was new to the story, he had to be brought up to speed on the requirements, design and development done on the story. This led a seemingly simple story to span across different sprints and almost mutate to a more complex story, enduring scope and personnel change.

This split of the story affected the overall shape of the work in both the sprints, both for Natalie and for the team at large.

4.4.6.8 Changes in participation over time

As we see in Figure 4.12 on 103, our participant observer Natalie, was working on the maintenance team and this phase also included her onboarding time. Initially, Natalie and Nathan worked with Davin, and through the rest of time on the maintenance team, she primarily worked with Nathan, the other newcomer. Natalie would occasionally consult with other pioneers or early onboarders on different bugs, but the bulk of their programming was done either alone or in paired programming sessions together. Natalie would also be reporting to Philip during the daily stand up meetings. The work on each bug would last anywhere between one to three days. As the maintenance team was also responsible for ensuring the daily builds do not have problems, Natalie would often work with Moss, the infrastructure personnel to remedy any software build related issues. As the figure suggests, Natalie's interaction was primarily with Nathan and Philip with several consults with pioneers like Alex or early onboarders like Davin or Ivan.

After a long stint on the maintenance team, Natalie and Nathan moved to the Pre-release team, where they were joined by pioneers like Alex and Quinn and early onboarders like Karoline, Ivan and Calvin and the third newcomer, Casey. As we see in Figure 4.13 on page 104, Natalie worked primarily with Alex and Karoline on different long running bugs together and on some short running bugs by herself. As we see in the time line for the first three weeks of the pre-release work in Figure 4.9 on page 100, Natalie worked on some bugs that ran as long as medium sized user stories did. During the pre-release work, the team met daily with Steve, who was the Scrum Master and with Douglas who was the Product Owner and the customer proxy on many stories. As many of the bugs that Natalie worked on had to do with infrastructure changes, she also extensively consulted with Moss on infrastructure matters.

When the product major version release finally took place, the teams were reconfigured again to form scrum teams. We observe who Natalie interacted with during the first sprint in Figure 4.14 on page 105. All her programming interaction was with Alex and Karoline, and during the sprint, Alex, Karoline and Natalie worked on two long running stories. There was daily interaction with Steve, Douglas and the rest of team during the stand up meeting and the demo prep before the end of the sprint. The second story, which was developed by Natalie and Karoline was almost complete and even given approval by the customer after demo, but was rolled over into the second sprint was the final code commit. This affected how the second sprint was shaped.

We see in Figure 4.15 on page 106 which describes who Natalie communicated with during sprint two, that her interaction was heavily with Calvin. Karoline and the customer proxy for the rolled over story were on leave for the beginning of the second sprint, so Calvin took

Karoline's place on the story and a new customer proxy was appointed. During the demo, the new customer described additional scenarios for the functionality to work, that the first customer had not mentioned. This expanded the scope of the story. So Natalie spent some time with Calvin, bringing him up to speed on the story. As the base functionality was now working, extensive testing could be performed. Natalie then worked with Jenna, who was a new tester working with the team. As Jenna was also relatively new to the system, Natalie spent some time helping Jenna get accustomed to the new functionality and helped with setting up tests for it. Then Karoline, Leon and Natalie worked on a long story, where as the pioneer and expert in the functionality being changed in the story, Leon led the design effort on the story. Karoline and Natalie worked on different parts of the story individually, as dictated by Leon. This changed feature also required extensive testing, leading Natalie to work with Jenna some more. As with the first sprint, the development team met with Steve and Douglas daily for stand up meetings and towards the end of the sprint for demo planning and preparation.

Figure 4.16 on 107 depicts Natalie's communication and interaction during sprint three. We observe that Natalie's interaction during this sprint was a lot less intense than prior sprints. As the stories during this sprint were related to stories already completed in prior sprints, the work in this sprint was largely performed in solo programming sessions with brief communication bursts to reconfirm and tweak design decisions that were established before. At this point, all the developers on the stories knew what they had to do, and the development effort was smooth, requiring limited interaction. Natalie pair programmed with Casey for a short time.

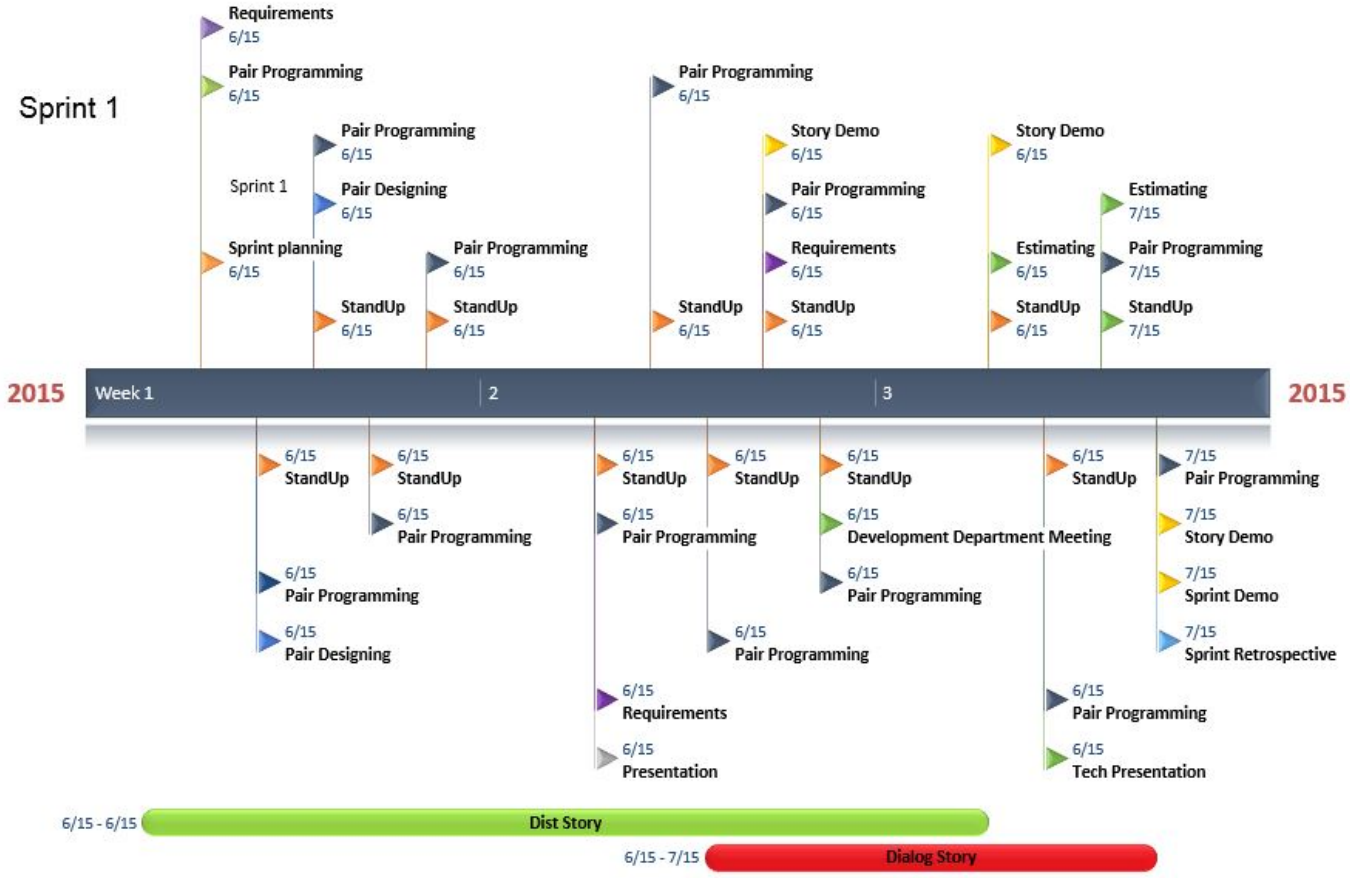

Figure 4.8: Sprint 1 Timeline.

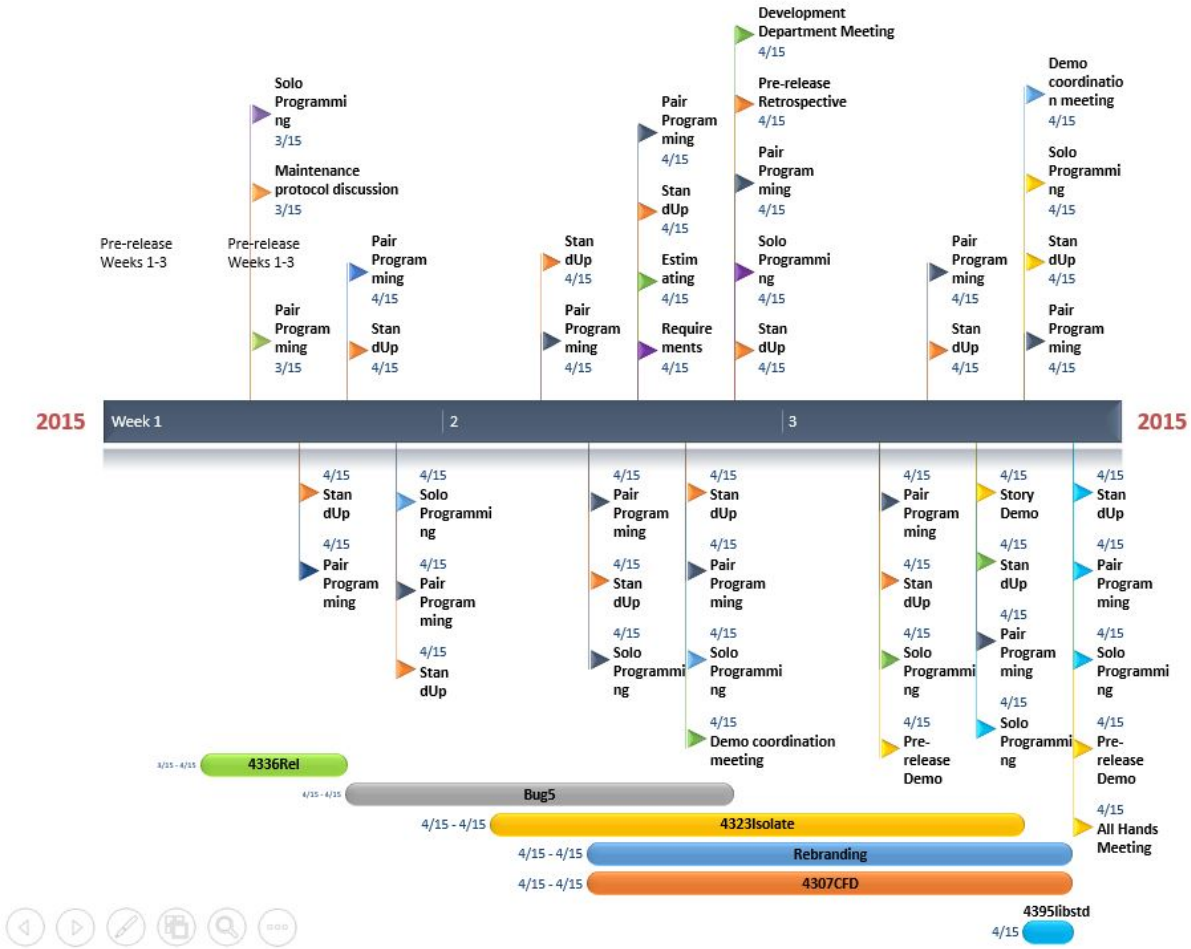

Figure 4.9: Pre-release Weeks 1-3 Timeline.

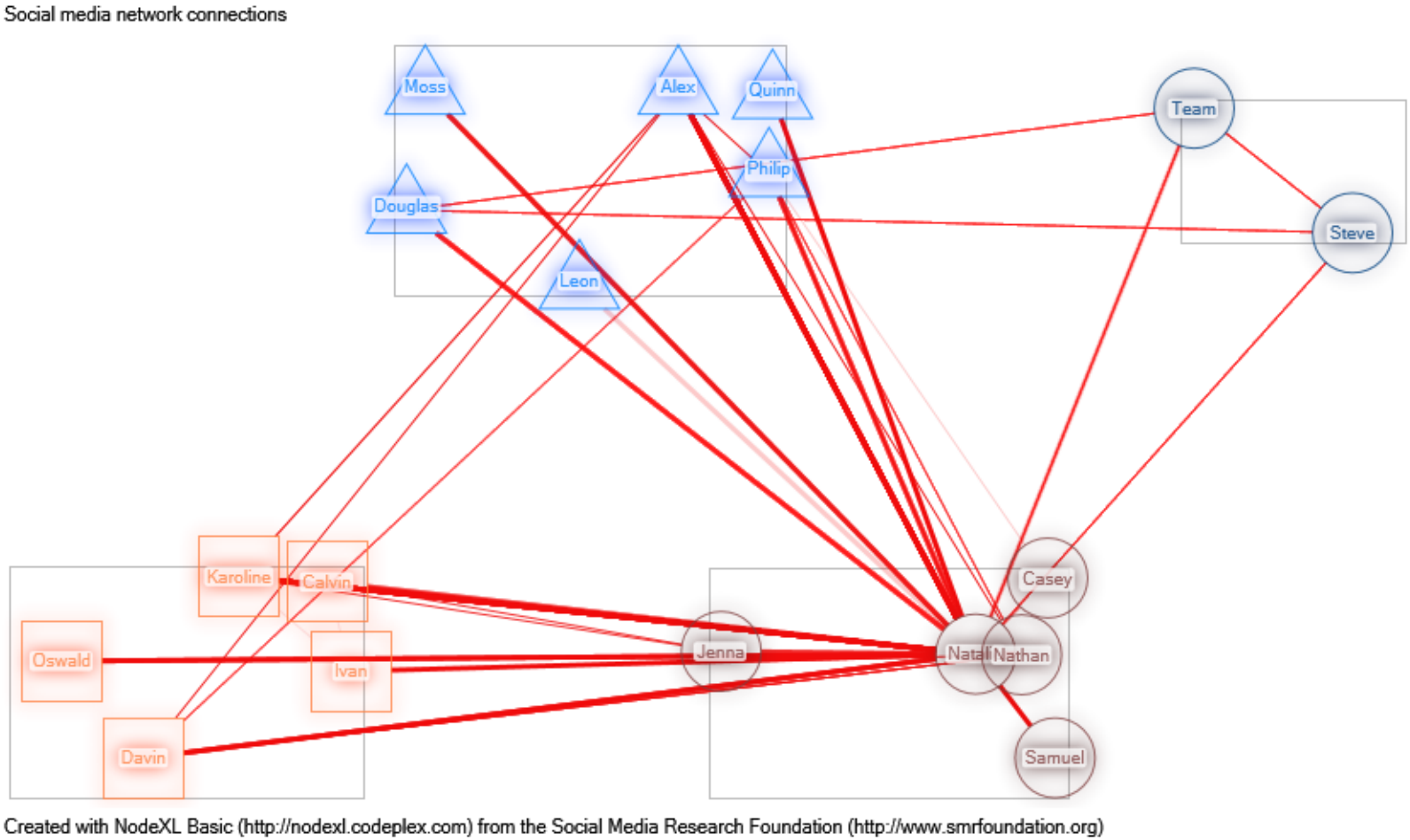

Figure 4.10: Fruchterman-Reingold with groupings for all interaction at TAI.

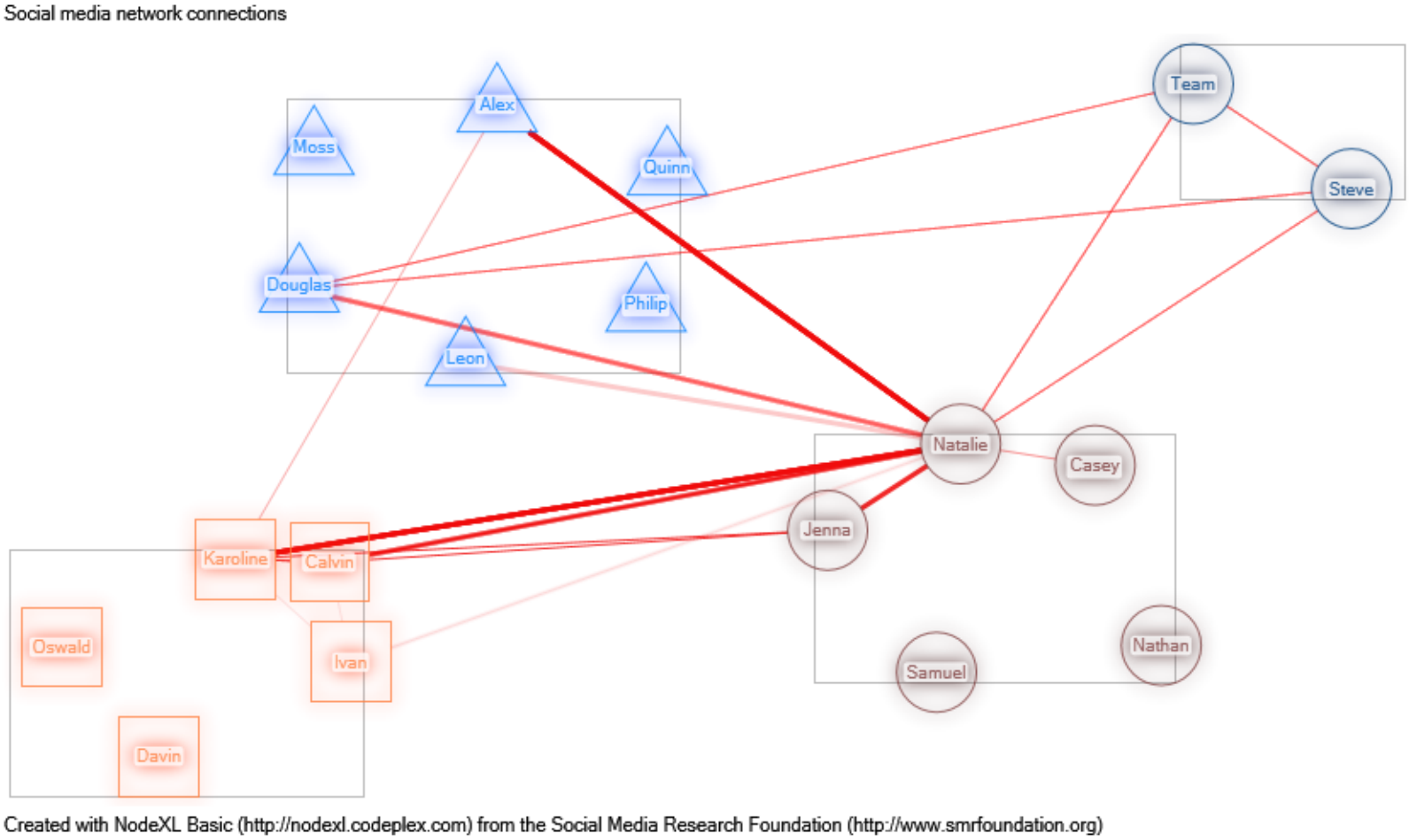

Figure 4.11: Fruchterman-Reingold with groupings for all interaction at TAI.

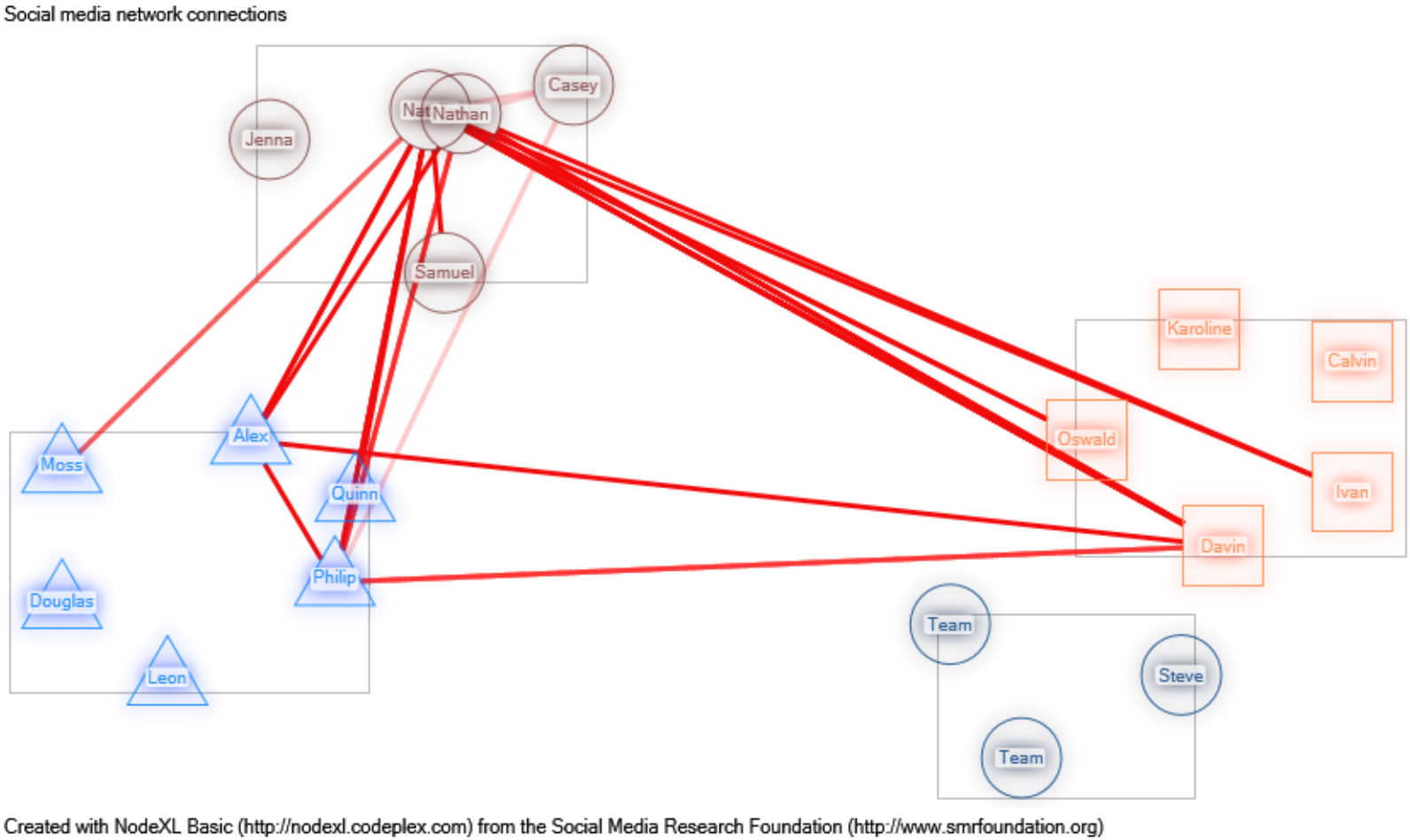

Figure 4.12: Fruchterman-Reingold with groupings for all the Maintenance team.

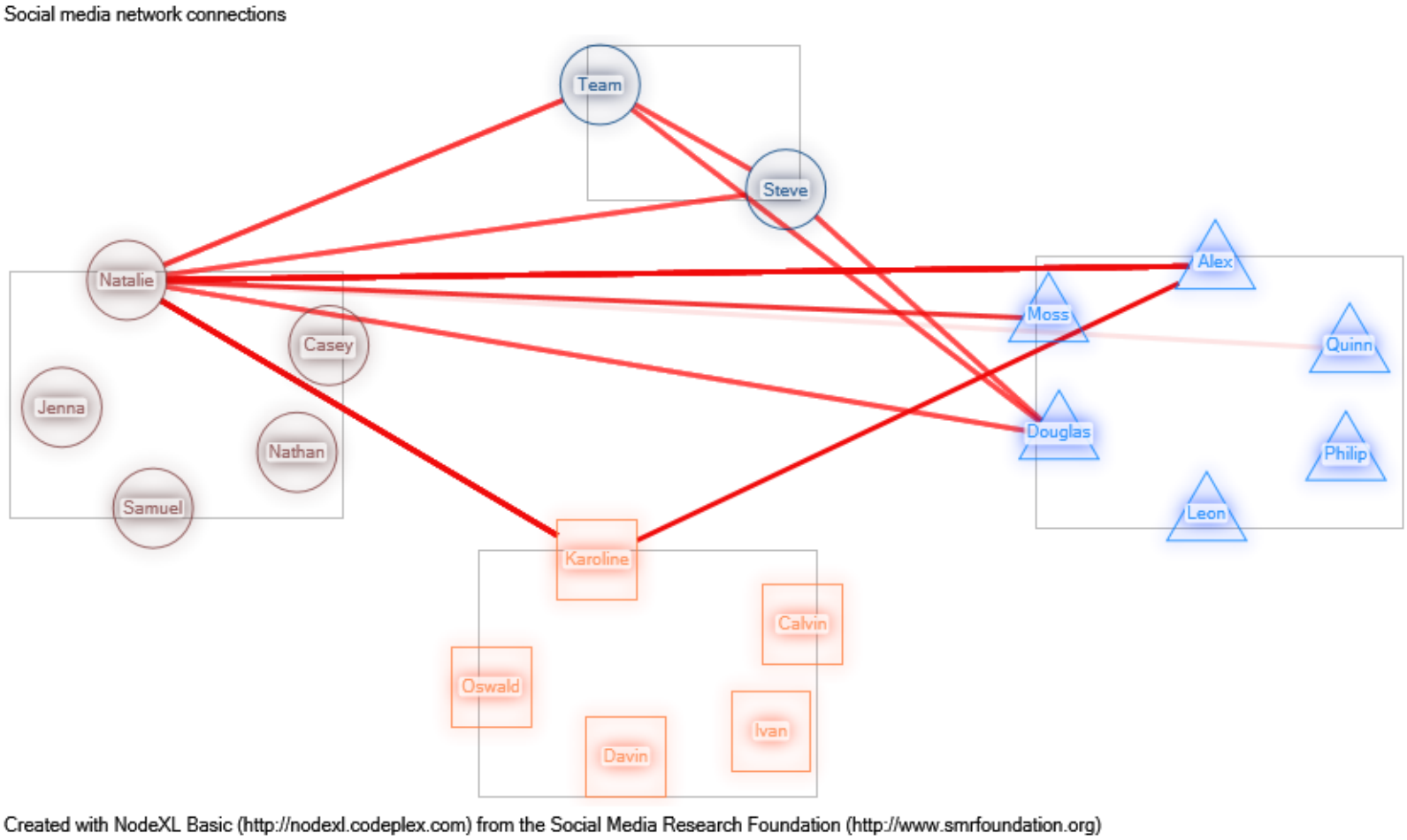

Figure 4.13: Fruchterman-Reingold with groupings for all the Pre-release team.

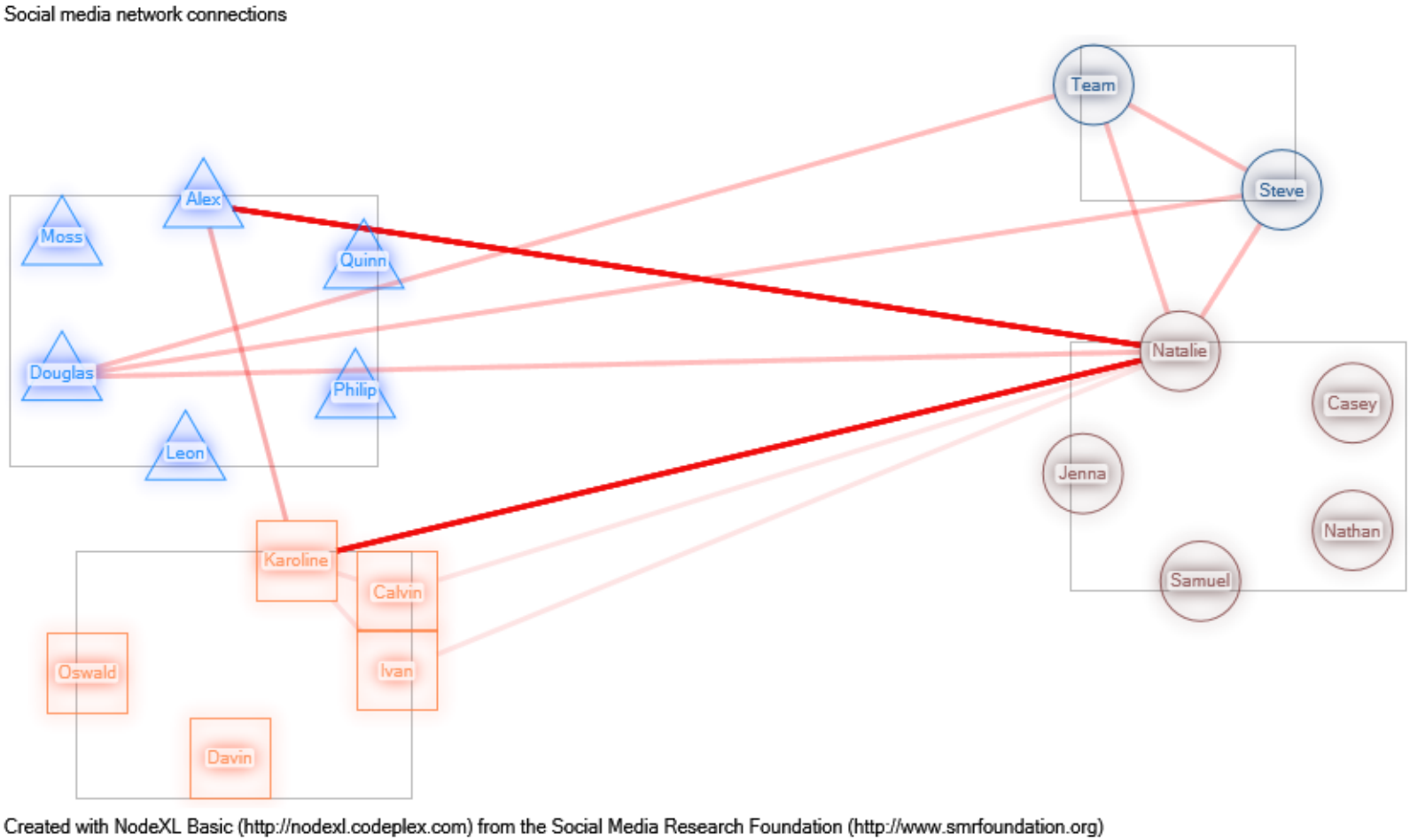

Figure 4.14: Fruchterman-Reingold with groupings for all the first sprint cycle.

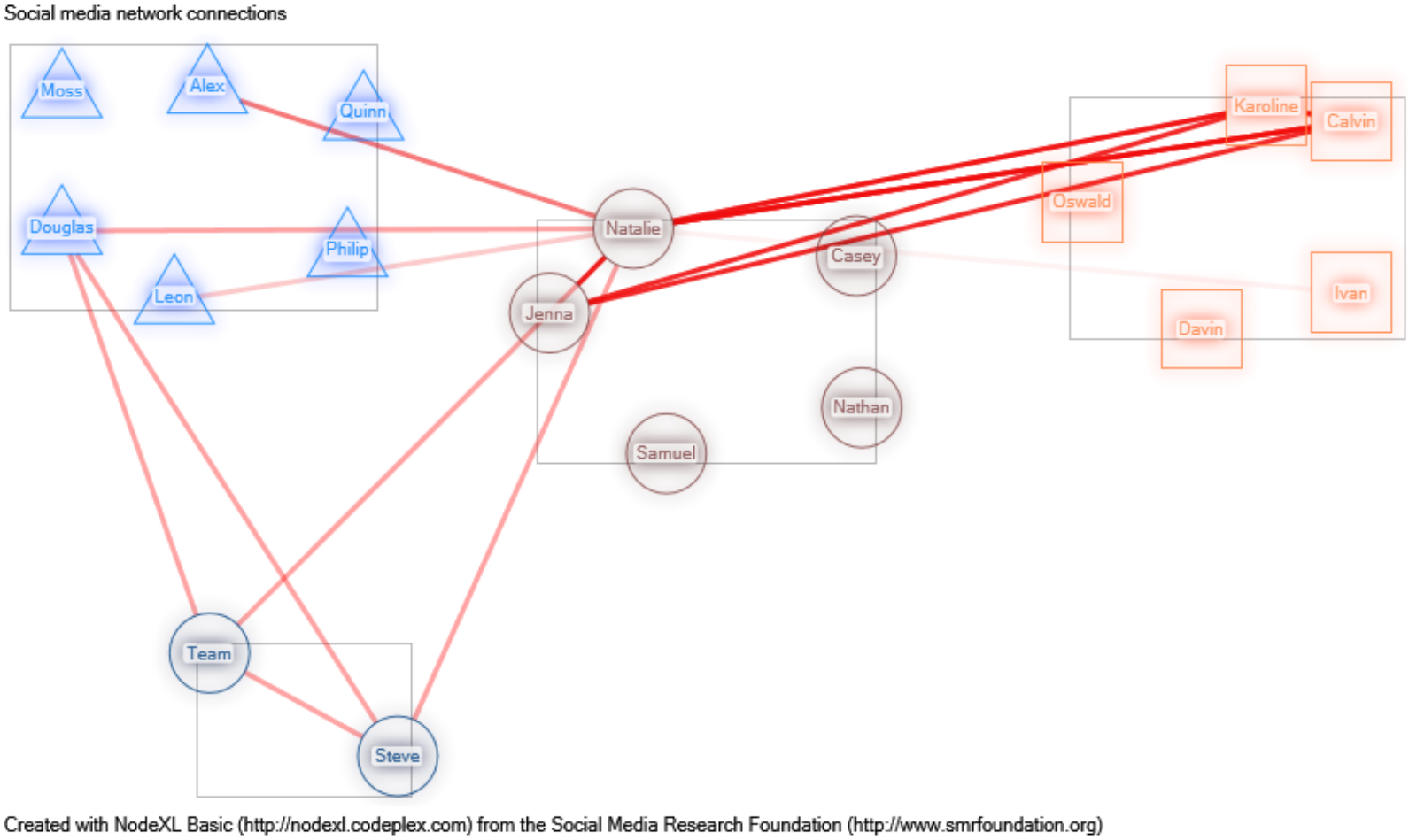

Figure 4.15: Fruchterman-Reingold with groupings for all the second sprint cycle.

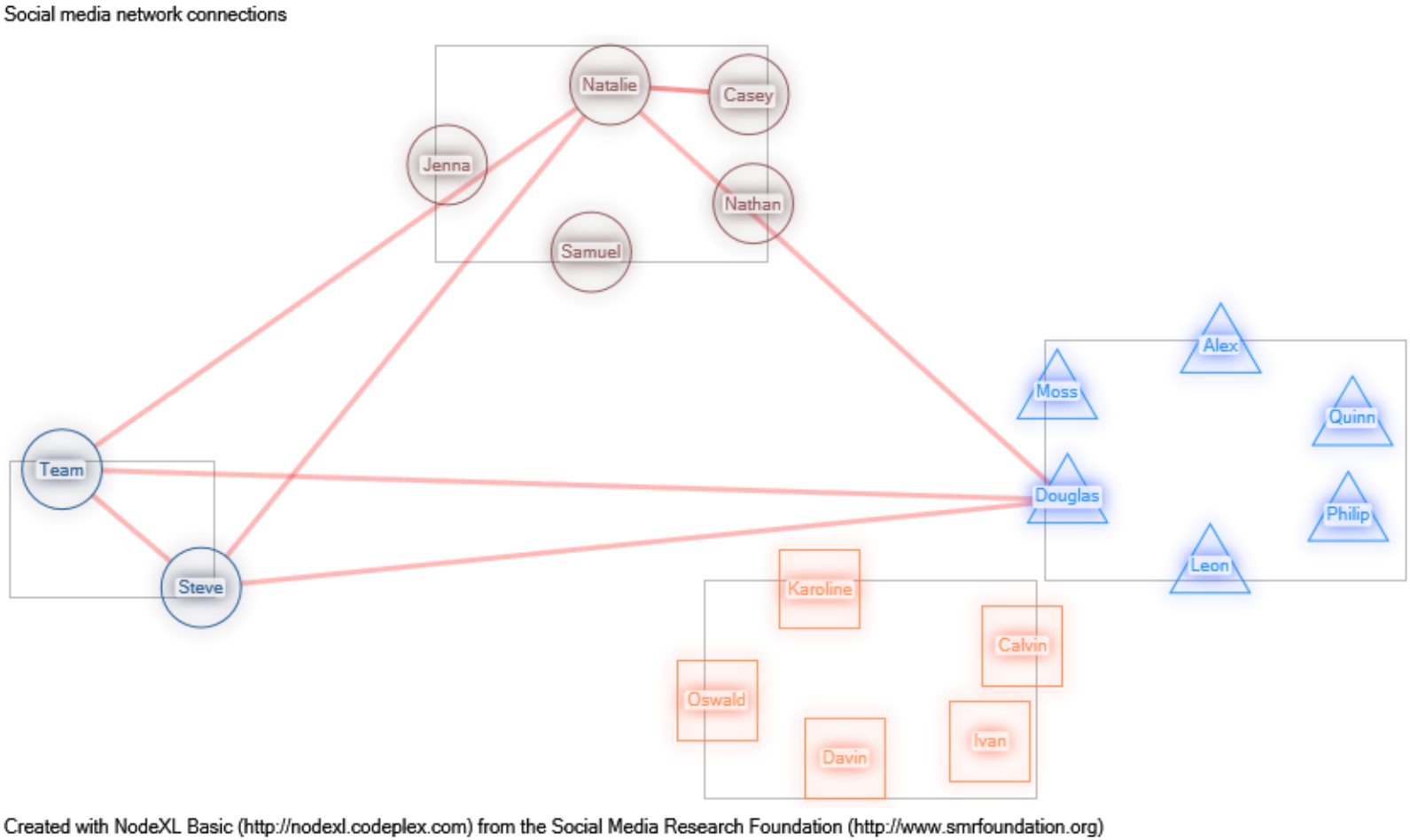

Figure 4.16: Fruchterman-Reingold with groupings for all the third sprint cycle.

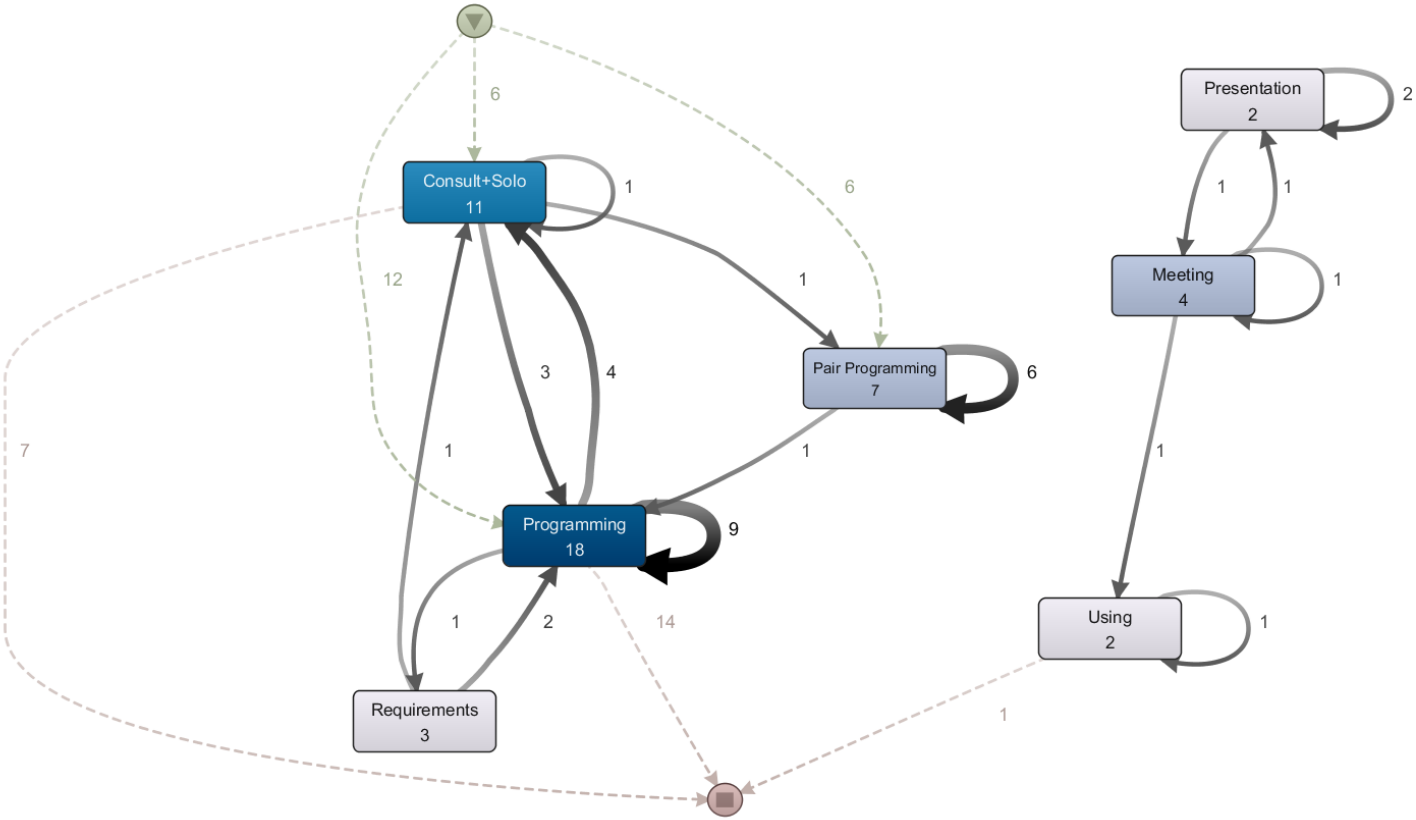

Figure 4.17: Per bug process filtered to a 50% activities and 50% of paths for Maintenance Team work, output from process mining tool Disco

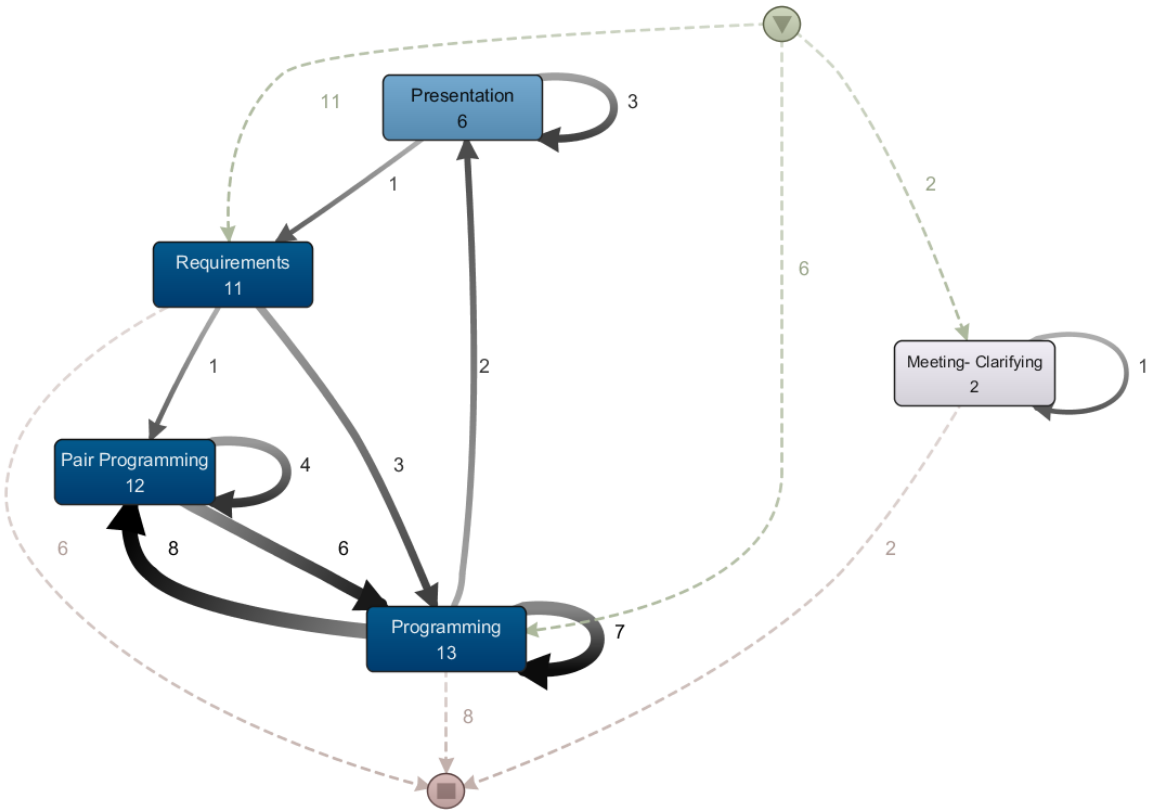

Figure 4.18: Per bug process filtered to a 50% activities and 50% of paths for Prerelease Team work, output from process mining tool Disco

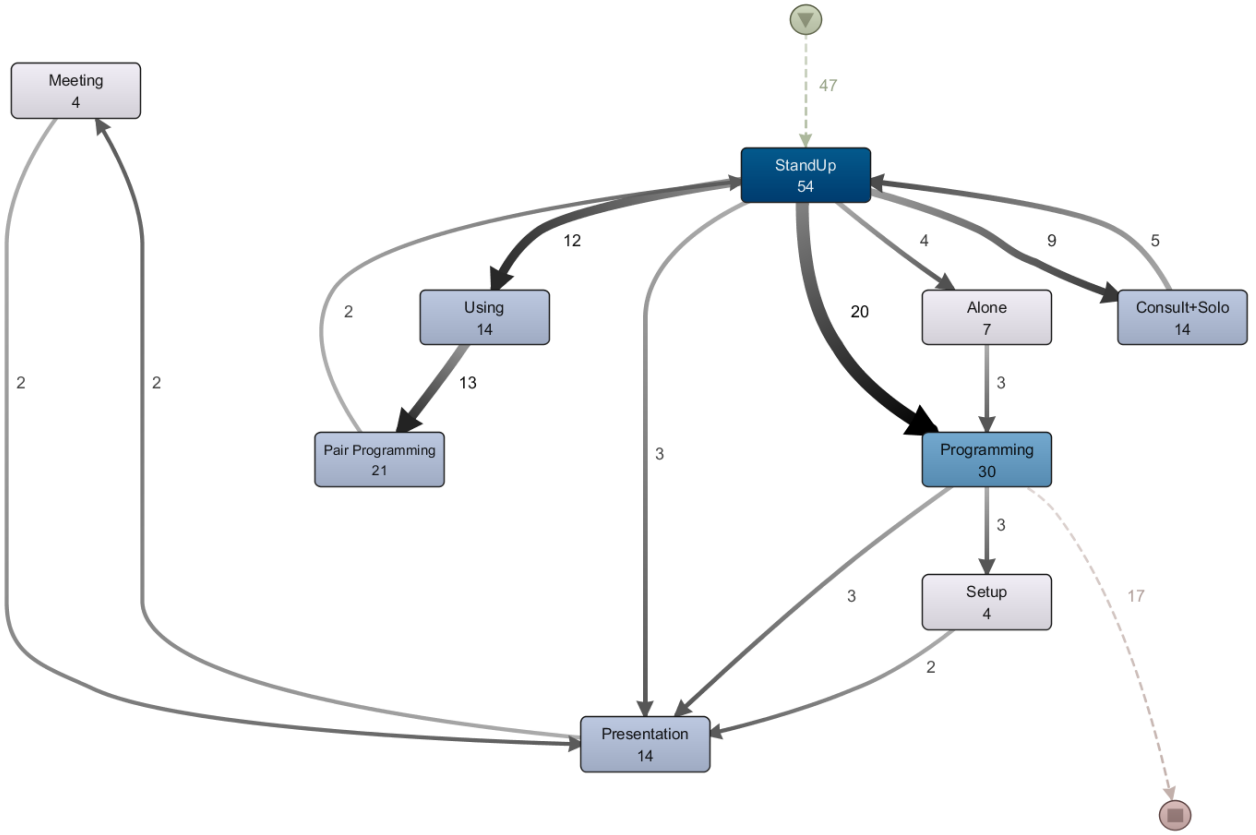

Figure 4.19: Per bug process filtered to a 50% activities and 0% of paths for Maintenance work, output from process mining tool Disco

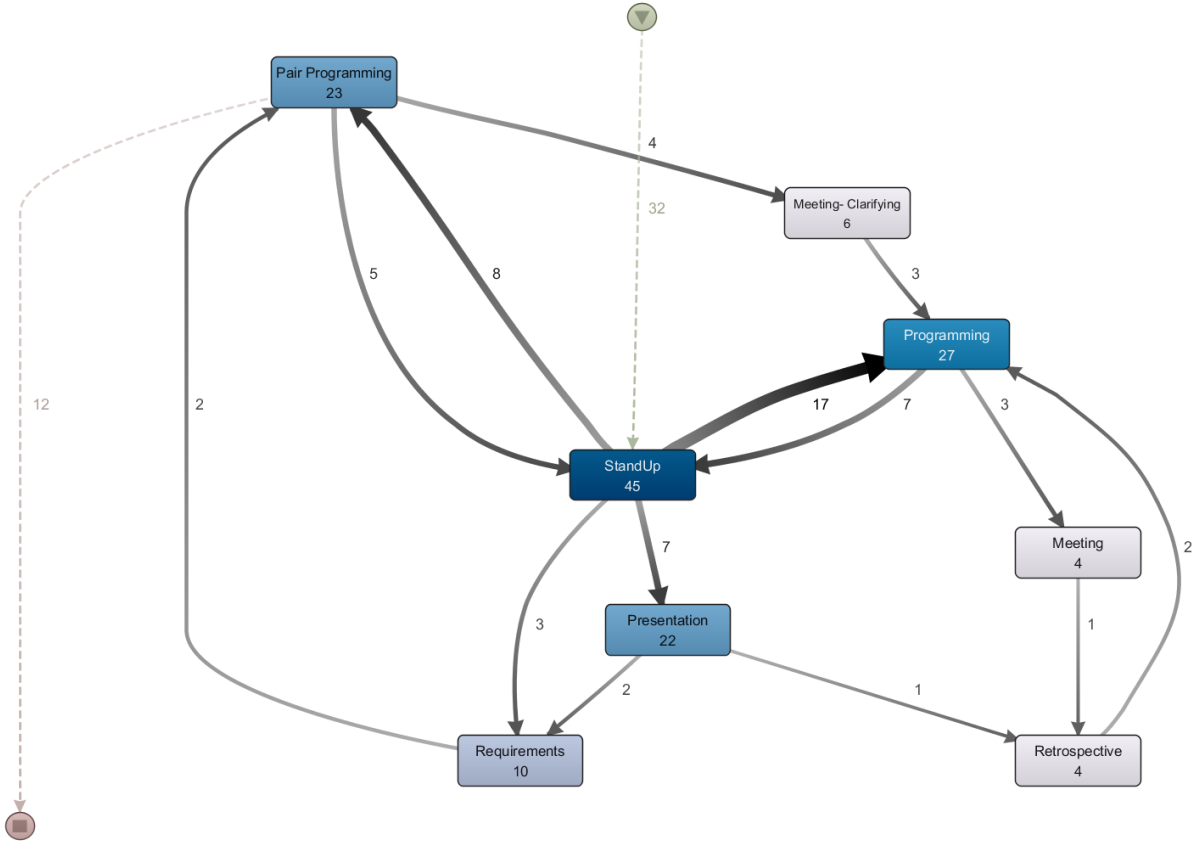

Figure 4.20: Per day process filtered to a 80% activities and 50% of paths for Prerelease work, output from process mining tool Disco

Chapter 5

TAI - Results

From our study of TAI, we used qualitative data from the developers interviews and the observations and reflections of the participant observer to form more in depth observations and uncover the story, patterns and the social, historical and cultural context. We then used the quantitative data to validate and ratify some of our observations.

The greater purpose of our study was to capture patterns of communication related to mentoring and knowledge and process management that allow software development communities of practice to evolve their practices, sustaining themselves through change. In a true grounded theory sense, part of the job was to ask the question: 'what are the questions?' The search for patterns is in fact also the search for the questions. Our study of the software development community of practice has a broad focus and research questions. Broader studies help capture different aspects of the subjects, while allowing researchers to see many facets at play together to assess what the important and unique aspects of the subjects and their relationships are. A purpose of our study was to identify potential questions for future research.

In this chapter, we discuss some of our findings based on our mix of quantitative and qualitative analysis and present the results of our findings in the form of patterns.

5.1 Discussion

In this section, we discuss some of our observations from the mix of quantitative and qualitative data, but with more context than observations in the previous chapter. Some of these

discussions will allow us to identify and frame our findings as patterns in later sections.

5.1.1 Types of community participants

Based on the categorization of communities of practice participants by Wenger et. al in the their book Cultivating Communities of Practice [68], we observe that the core group of the community of practice consists of Philip, the software engineering department head and most enthusiastic process supporter, and Alex, who is one of the pioneer developers who is still actively involved in the primary development effort, working with other pioneers, early onboarders and newcomers combined, Quinn, who is not an original pioneer but with over a decade of experience on the same code base, is effectively a pioneer as far as knowledge and influence is concerned. The core group explicitly plans and encourages knowledge management within and outside the community. Philip is the coordinator of the community of practice and actively builds strategies for knowledge sharing, development and management, imposes practices, decides to adopt systems, and experiments with new formats of communication.

Some early onboarders and most newcomers form the active group, which engages in the community of practice most often and perhaps benefit most from the community as well. Some examples of such regularly engaged early onboarders are Karoline, Ivan and Moss. Under Philip's directive, Ivan and Moss also help organize explicit knowledge sharing sessions called Tech Talks within the community to share and discuss a variety of topics related to software development and what it means to them.

There are other developers, who adopt peripheral participation where they are part of the community of practice and attend and learn from it, but may not regularly contribute to it. Some developers who work on niche development topics or projects that do not have a lot of overlap with the work other developers are doing and benefit only from occasional participation.

There are some outsiders to the community, like the mechanical engineers and system modelers who rarely participate within the community of developers, however, there is knowledge that flows bidirectionally between the two groups. However, their interaction is rare. As the coordinator, Philip has been trying to convince the modelers team to interact more often with the developers, and learn more about the new features of the product, so that they may use them and give early feedback, and also be able to advise clients.

5.1.2 Community of practice evolution over time

We observe that the community of software development practice at TAI evolved over time and changed in format and recognition as a community by the organization. We assess the community over time based on the classification by Wenger et. al in their book on *Cultivating Communities of Practice* [68].

The software development community of practice at TAI went through many transitions over time. Initially, when there were very few developers, and they all worked separately on massive projects, they consulted with one another and reviewed each others' code during commit time. They were a very small community of developers then and many of them did not consider themselves programmers as the background for many of them was in mechanical engineering. However, as a small community, they enabled their collective knowledge growth and created the combination of physics and mechanical engineering background with programming skills, giving this small community a unique flavor, forming the core of the business at TAI. At this point the community was largely 'unrecognized,' as even the members barely recognized themselves as a community and their identities were still mixed, as many considered themselves mechanical engineers who are programming. Other than code review, there was no other formal practice.

Over the next few years, the focus of the company slowly shifted to a mix of small projects for different customers and a primary code base from which some similar products emerged. This required a combination of knowledge of the code base, the primary product line, the physics behind the different small projects and a means to integrate both worlds. This requirement of a new knowledge profile caused the programmers, mechanical engineers and physicists to work more closely than before and share knowledge of different types. This was the mode of operation for many years at TAI and this knowledge sharing based community of developers expanded. This community was recognized by the organization and some means of software process started to be adopted. The development effort was starting to be larger than could be managed in the typical one developer to one large project ratio. Developers needed to work together and incorporate concepts from physics and mathematics into features of the code base. The addition of practices and conscious decision making about software process, even though the process would change over time, meant that the organization acknowledged that this group of developers had a need to work together regularly and share knowledge to accomplish their work and grow in capability. The community was now 'supported.'

There was a change in the structure of the software development, when Philip became the team leader for the software development effort. Philip notes that the software development at TAI felt like using a mechanical engineering process and trying to apply that to software,

and it just did not fit. The community of developers could use practices that suited their nature of work better. Philip worked with Alex to learn about practices and processes that the software industry employs. They worked together to identify a particular set of software practices and decided to adopt Agile methodologies, in particular Scrum for their growing development team needs. The practices of the process dictate when and how often to communicate and what to communicate. They also adopted some aspects of pair programming. They continued with their earlier well proven practices of code review. At this point, the community of software developers was 'officially recognized' as an entity with needs of knowledge and expertise management different from the rest of the organization. It was also recognized that the community can manage itself and decide for itself the practices and tools it adopts to get the immediate work done, but also to sustain itself over time as developers change roles and projects and as new members join and come up to speed. In some ways, the community was also starting to appear 'institutionalized'.

After operating in this organized mode for a few more years, the software development effort saw a lot of growth in the number of developers employed, and in the varied types of work being done by the developers. The main product line also became a prominent source of income for the company, in addition to its government and industry funded research and services projects. As the department grew, the code base also grew manifold. The variety of product and code features and their applicability to different types of clients and end users grew as well. It was starting to become untenable to operate efficiently with the existing structure. Again, Philip, with input from Alex and many other sources like successful examples of industry practices in software development and some input from Steve, determined the knowledge hierarchy followed by the company Spotify would be a good model to adopt. So we see another big change in the organization of the software development department to facilitate knowledge growth and sharing. This model describes a changed reporting hierarchy, to allow developers to work on their projects but also grow and hone their skills in their chosen areas of expertise. This new setup allows for more deliberate skills and expertise management, built right into the institutional reporting structure. At this stage, this community of practice is fully 'institutionalized.'

5.1.3 Different onboarding strategies

We observed different deliberate onboarding strategies at TAI. When the current early onboarders first joined the development team, they typically started with low impact work like working on bugs in the Maintenance team using Kanban, but paired extensively with knowledgeable pioneers to gain a lot of knowledge fast.

For the current newcomers, two onboarding trajectories have been attempted that are strategically different to explore onboarding methods. The most effective strategy used most commonly in the last few years is the *Newcomers with early onboarders* strategy where newcomers initially start with low impact work in resolving bugs in the maintenance team following kanban, then move to high impact work in sprints following scrum, working with a mix of early onboarders and pioneers. The second is the *Newcomers with pioneers* strategy where newcomers start right away with high impact sprint scrum work, working closely with a pioneer, getting one on one time and eventually moving on to work with early onboarders on lower impact work. The latter strategy is similar to what early onboarders experienced when they first joined but has not remained the predominant strategy. It was tried with Casey after a long time.

5.1.4 Different formats of communication

As a community of practice, the most important currency combination is knowledge and communication. There are different types of knowledge - explicit and implicit and public and private. Then the means of communication used to share knowledge can be formal or informal, structured or unstructured and planned or unplanned.

We see several forms of knowledge sharing communication means employed in this software development community of practice. The formal and relatively public channel of communication was the *wiki*. The knowledge wiki ranged from specific information about decisions made for past and current stories, to what was delivered in each release, to the details of the information discovery process for most user stories and bugs. The wiki also included pages on generic topics like instructions for setting up new machines, or instructions for accessing and using some specific sort of software, etc. Some developers maintained pages of assorted bits of information that they had collected and shared with the other developers. The wiki was this dynamic, public, formal means of information sharing.

Another formal means of sharing information with peers was a series of presentations called Tech Talks. These were directed presentations that some developers organized to share knowledge on different topics, related to software development and software process. Developers also shared knowledge outside the developer community, with clients and product owners, engineers and other stakeholders in the form of regular *product demos*. Demos were typically held at the end of every sprint, or every week or so for maintenance work. Developers also received feedback on their work during these open sessions. There presentations are directed, formal and structured and based around sharing product specific knowledge.

Within the developer team, the daily stand up meeting was a ritual used to share status updates on their work. These meetings are also formal and structured, but are meant to be quick opportunities to touch base with the team. These meetings are about sharing updates and planning information.

Developers work together on the same workstation, programming as a team. A lot of information is shared during the process. This is also a means for knowledge creation. The most amount of knowledge exchange happens when developers work together.

Many other formats are employed ad hoc. The open conference model was occasionally employed and experimented with for brainstorming on major code decisions.

5.1.5 Knowledge silo management

One of the challenges that the development effort at TAI faces is sharing and spreading knowledge from the 'pioneers' to the early onboarders and the newcomers to the software development community of practice. All the early onboarders and newcomers were capable programmers and designers, but what made them different from the pioneers was that they did not have the full historical context for design choices and rationale across the entire, relatively large and long running code base. Conscious and concerted efforts were being made by the company to disseminate the information with other developers, allowing the pioneers to move on to work more appropriate for their skill set, removing dependency solely on them and to empower the next generation with the knowledge to keep it from being lost. There is documenting of design choices and questions that the client answered specific to design choices, but it is difficult to document the reason why every other possible design choice was not taken, even though this information may be useful at some possible future juncture. It is often this knowledge that is most valuable because newcomers cannot possess it and are largely dependent on the pioneers time in substantial quantities to learn it. The more knowledge a pioneer has, the more expensive and over extended their time and availability can tend to be, making this knowledge transfer that much more difficult. Capturing and preserving this implicit historical knowledge proves to be a challenge that a lot of software development communities face in sustaining themselves, especially in a culture of high attrition that plagues the industry. If we were to study the same workplace or other workplaces again, we would want to focus on and collect more details on how this knowledge sharing affects trajectory, participation and identity.

5.1.6 Learn communication style over time

The two main things a developer attempting to incorporate into a software development community of practice needs to learn are the communication norms of the team and the knowledge of the code base and understanding code style norms and programming practices, along with the historical context of design rationale for the interrelations between different parts of code. Most developers are capable of over time, discovering the relationships between different parts of code on their own and learn the code style and programming practice with exposure to the code. They are often dependent on documentation and mentors and colleagues to learn about the design history and rationale. And they learn the communication practices and conventions through exposure and instruction. Recurring, scheduled communication practices are learned relatively easily, however more subtle nuances in communication are more difficult to pick up. These can be learned over extended time or from the colleagues. At TAI, we observed an explicit acknowledgement from the pioneers and the managers of the development department of the importance of the communication practices. That culture seeps through and other developers also follow and appreciate the importance of communication to the software process. This helps when newcomers are learning the ropes as other developers demonstrate and teach them about different communication practice.

5.2 Previously Discovered Patterns at TAI

From the existing library of Scrum Patterns [20] and Organizational patterns [22], which together forms intersecting pattern languages, we observe the occurrence of many patterns in the operation at TAI.

5.2.1 Previously discovered mentoring patterns

The *Domain Expertise in Roles* pattern [22] talks about hiring experts to form a team, as this approach makes the most effective teams. One risk of this pattern is that the roles can get too narrow and cause the organization to be dependent on a few individuals. In the first decade or so of its existence, this was the state of the organization and role breakdown at TAI. All developers were experts in their area, but this also made the organization extremely dependent on them and have their ability to function impeded by an absence or any attrition. One approach to implement this pattern is by training newcomers really fast

and well to operate at expert level more quickly. This allows for expertise to be shared and allows the roles to remain important but not irreplaceable. This is in line with the *Moderate Truck Number* [22] pattern which talks about avoiding a situation where just a few people have everything dependent on them. This patterns talks about encouraging a culture of knowledge sharing to spread responsibility across many different people rather than having a few people do everything.

We observe that for knowledge management and mentoring, the software development community of practice at TAI adopts the *Apprenticeship* pattern [22] which uses apprenticeship to convert newbies to experts fast. We observe that no matter which form of onboarding and mentoring they adopt- onboarding by pioneers or by early onboarders, the format followed is apprenticeship.

When the early onboarders first joined, almost all the experts would spend time with them, teaching them different aspects of the development effort that they wuld require. A problem with this approach is that the more newcomers join, more of the experts get tied up catering to the onboarding and mentoring effort, the scenario described in the *Day Care* pattern, where experts spend a lot of time babysitting the newcomers, and the pace of the overall development effort is hampered. The solution in the Day Care pattern advises assigning one expert to take care of the newcomers so the other experts can continue working without being distracted by the newcomers.

Both the Apprenticeship and Day Care patterns are governed by the *Old People Everywhere* pattern [3], pg 215 which emphasizes a need for the young or less experienced developers to interact with the older or more experienced developers, which is better for the novices, the experienced builders and the community as a whole.

5.2.2 Previously discovered roles

We observe some role patterns at TAI. Alex is the pioneer who is most actively involved in the main software development work. He leads the design and estimation work in most cases and is almost always involved when anything significant about the core functionality is changing. He advises the product owners and team managers on team organization and product management. Alex's position and influence in the team makes him indispensable. However, it also makes him a bottleneck to progress as he is a required resource or consultant to everything. This was manageable when the work in the company was limited, in the early days. However, now with all the different products and projects across different teams, Alex was spread too thin. He fits the *Legend Role* pattern [22] where one person becomes indispensable. The pattern talks about naming the role after that person, so that

others who have to take on the legend's responsibilities are aware of the composite nature of the work, and do not assume limited scope from a narrow or ambiguous title. The pattern talks about how it is necessary to provide training to those taking on the role and responsibilities after the legend, and it should be the legend who provides that training. The pattern also says that the legend's role fades over time as others take on subsets of the legend's responsibilities.

We see that Philip fits into the *Patron Role* pattern [22], pg. 133, who allows for continuity of the project by serving as a champion for the project as a visible but accessible manager, while allowing for the development team to manage itself to some granularity.

As we notice in the story where Karoline was part of a team where some younger developer would find it difficult to work together on a team as the younger developer's tone would sometimes be unintentionally harsh or arrogant. This was making all the developers on the team uncomfortable. Philip reshuffled the teams and brought in Gerome, who fit the pattern of the *Peacemaker Role* and helped keep the temperament of the team calm and collected and returned interactions to smooth running.

As the organizational patterns dictate, it helps to hire a specific person for the pattern of *Mercenary Analyst*, to handle all the technical documentation and user guide, and ensure they are kept up to date with code base changes. TAI employs a dedicated technical writer who works closely with the development teams and is a permanent member of the Scrum teams to keep on top of documentation changes.

The development team at TAI also employs the *Surrogate Customer* pattern where they use internal resources to serve as customer proxies and represent the customer's interests. There is a customer or a customer proxy assigned to every story.

5.2.3 Previously discovered modes of operation

The *Diverse Groups* pattern talks about how groups with diversity in temperament and ability are more effective, however patterns like *Self Selecting* within teams tends to encourage a more homogeneous team to be formed. At TAI, initially the number of developers was small, and they all were men who came from a mechanical engineering background who were now programming. All these developers were roughly the same age, and had roughly the same level of experience as most of them started working at TAI after their undergraduate and graduate studies. Over time, as developers of different levels of experience and different backgrounds and temperaments are hired, the total number of developers increases and the potential for diversity among them as well. The management at TAI intentionally

shuffles its developers among the three to four different development teams to counteract the tendency towards homogeneity. The Scrum teams that Natalie worked on typically had one or two pioneers, two to three early onboarders and three to four newcomers. There was also greater gender and age diversity in the team.

We observe that in the Scrum team where Karoline and Ivan worked together, there was occasionally an issue with Ivan taking on multiple tasks, potentially causing himself to become a bottleneck to progress, something Karoline disagreed with in practice. However, when Alex and Calvin are on the Scrum team with Ivan and Karoline, they ensure that they figure out which tasks to team up on and prioritize so that the entire team collectively is able to function. If someone is likely to not finish a task soon or is taking on a task that others depend on, some members of the team lead the discussion to decide how to manage task allocation to allow the flow of work to continue. This is in line with the *Someone Always Makes Progress* pattern, which talks about ensuring that the team keeps moving forward and making progress on the primary task.

The standard mode of programming at TAI used to be working by oneself on a big project. Over time that practice changed to become the *Develop In Pairs* pattern of programming. Within the Scrum format, the team self selects to assign different user stories, which often describe features, to themselves. So the developers working on the same user stories, form a temporary sub-team. This is in line with the *Feature Assignment* pattern. There are times that during the feature development or user story work, other distracting tasks come up, like investigating the cause of test failures on their feature development branch or the primary product development branch of the code base. This can potentially derail the primary development effort. The development teams at TAI sometimes adopt the *Sacrifice One Person* pattern, where one person takes on the distraction task and the remainder of the team continues uninterrupted on their development work.

The daily task allocation and work management is usually done by the team, where the discussion for this type of allocation is led by the Scrum Master. This allows the team to follow the *Informal Labor Plan* pattern where the developers devise their own short term plan. Using the *Sprint Planning* pattern, the team participates in deciding relatively more long term operations. As the team self selects and takes turns doing so, they are able to follow the *Distribute Work Evenly* pattern.

Occasionally, the development department manager will pull all the experts together to brainstorm for a solution to some specific issue or new endeavor. He will allocate a significant chunk of time, like a few continuous hours for the team to work on the issue and come up with different strategies from their discussion, thereby employing the *Lock Em Up Together* pattern.

Recently, TAI has put in motion a plan to reorganize the software development department where there is explicit grouping of developers based on their existing and intended skill and knowledge growth, employing the *Subsystem By Skill* pattern.

5.3 Novel Pattern results

Our objective is to build on the existing body of pattern knowledge in Scrum patterns and Organizational patterns, as our studied scenarios fall under either category or at their intersection.

To present our results, we adopt an amalgam of the common template used by Scrum and Organizational patterns. The two communities already have a common template to allow interplay between the two sets.

The template, represented below, covers a pattern name, a description of the problem and the solution, along with a rationale and resulting context section that describes the result and risks of using that pattern.

PatternName

Problem Statement

Therefore: Gist of the solution

Resulting context and Rationale

5.3.1 Pattern: Pioneer Identity

Pioneer identity: Greater centrality to CoP and ownership of the code base, design and product. Possibly ownership due to actually creating it.

The Problem: How does a software development team harness knowledge and experience to its advantage.

Therefore: A team of developers works better when it is able to appropriately harness the

knowledge and guidance of its most experienced and most dynamic developers.

Resulting context and Rationale: Similar to the advantage of having a *Legend Role* pattern [22], having pioneer developers, who have extensive experience and knowledge of the code base, product features and historical design decisions.

5.3.2 Pattern: Early onboarder identity

Early onboarder identity Early onboarders feel like their identity has less centrality to CoP and ownership of the code base, design and product than pioneers. They feel ownership of certain components of the code base and product features that they personally created. They have participated in the design decision making process enough to have strong preferences, which may conflict with the preferences of pioneers.

The Problem: In a sustainable community of practice, it is not possible to have all experienced experts or Pioneer Identity developers. The community needs to have a means to sustain knowledge and growth.

Therefore: It helps to have an organized and concerted effort to include new developers into the community, who are trained and learning directly from the expensive pioneer developers.

Resulting context and Rationale: The early onboarders join the development team when it has primarily pioneer developers, but the team recognizes the need to grow and share knowledge to allow the community to sustain and evolve and also allowing the pioneers to move on to other roles.

5.3.3 Pattern: Newcomer Identity

Newcomer identity:

Least amount of ownership and centrality to CoP. May feel some familiarity with the code base over time, but not with the product, as they may not be users of the product and may not have enough exposure to users and customers yet to understand the rationalization of priorities of product features and use.

The Problem: In a productive, sustainable and steadily growing software development community of practice, how is growth maintained.

Therefore: It helps to have an influx of newcomers into a growing software development community of practice, where the newcomers are onboarded over time to become fully productive.

Resulting context and Rationale: The newcomers who are onboarded and trained by a mix of the early onboarders and pioneers help expand the abilities of the community of practice, allowing it to evolve. Their addition also helps increase the centrality that early onboarder identity developers might feel, as they switch from the junior mentees to a mentor role, increasing their centrality of identity.

5.3.4 Pattern: Pioneer Onboarding

Pioneer Onboarding: A new developer who has limited experience with Scrum and no experience with the product, enters a team of pioneer developers.

The Problem: In a team of pioneer developers (the developers who have the most extensive amount of experience with the code base), how do we bring on board a new developer, with no Scrum and product experience?

The pioneer developer's time is precious, and needs to be utilized efficiently to onboard the new developer.

The new developer needs one on one time with the pioneer developers, but also needs time alone to go through the code base and understand it. Finding a good combination of things developers can do on their own and with the pioneer developers is in the best interest of the team.

Therefore: Pioneer developers spend one on one time with new developer and explain the historical context of the design decisions on the code base.

New developers can go through product tutorials to understand how to use the product. And they read and edit the code alone, to understand how the different areas of the code are connected to each other. They also learn code style norms this way. New developers can go through system architecture descriptions and sequence diagrams to understand the connections and workflows.

Pioneer developers can pair program with new developers on tasks. Taking these pairing sessions as opportunities for interaction, pioneers can share the historical rationale behind major design choices.

Resulting context and Rationale: As a result of this mixed form of onboarding and mentoring, we use pioneer developers' time effectively and help new developers get up to speed on their roles. This initial investment by pioneer developers helps new developers become able to contribute towards the development effort sooner and more effectively. This allows pioneer developers to be aided by new developers, instead of just spending extended periods of time onboarding them.

5.3.5 Pattern: Generational Onboarding

Generational Onboarding : A new developer who has limited experience with Scrum and no experience with the product enters a team of developers with a mixed experience levels.

The Problem: In a team with a mix of pioneer developers (the developers who have the most extensive amount of experience with the code base) and medium level experience developers (early onboarders), how do we bring on board a new developer with no product experience by effectively using everyone's time?

The pioneer developers time is precious, and needs to be utilized efficiently. The pioneers have already spent time onboarding the early onboarders.

Therefore: Early onboarder developers spend one on one time with new developers for most of the time. The new developers occasionally consult with the pioneer developers to understand the historical context of the design decisions on the code base, on a case by case basis, as the new developers work on different tasks.

New developers can go through product tutorials to understand how to use the product. And they read and edit the code alone, to understand how the different areas of the code are connected to each other. They also learn code style norms this way. New developers can go through system architecture descriptions and sequence diagrams to understand the connections and workflows.

Early onboarder developers can pair program with new developers on tasks and work together for most of the onboarding. When the need arises, and when the pair encounters

a topic for which the early onboarders do not know the entire context, the new developers consult with the pioneers, to learn about the historical rationale behind major design choices, and accordingly make future design decisions.

Eventually, you work in a team that has highly experienced pioneer developers and other relatively less experienced developers.

Resulting context and Rationale: This mixed form of onboarding, mentoring and consulting uses the range of experience among the developers effectively to help bring new developers on board. We use pioneer developers' time only when their special expertise is needed and we get returns on the time invested by pioneers to onboard the once new and now medium experience level developer, when we use the medium level developers to onboard the newcomers. The medium level developers use their time to regularly pair with new developers, and share their knowledge of the topics they are most familiar with at that point. When their experience is found to be lacking in a particular subject, a pioneer is consulted. This also helps identify areas where the early onboarder developers still need to plug their gaps in knowledge, and allows the newcomers to build their initial knowledge base from multiple sources.

This format of onboarding may take slightly longer than the pioneer onboarding for new developers to be contributing at full capacity, but we accomplish a reasonably good onboarding without complete dependence on expensive resources like the pioneers.

5.3.6 Pattern: Process Tug of war

Process Tug of war: Within the same team, some developers are a stickler for process in preventing a bottleneck in development workflow and other developers may have a more flexible attitude towards task management.

The Problem: When there are differences of fidelity to strict adherence to process, how do you determine when to adhere to process and when to relax criterion.

Therefore: It helps to have the whole team weigh in on the decision to either enforce process or allow an exception and re-prioritize work accordingly.

Resulting context and Rationale: A risk of relaxing criteria often when encountering a disagreement about adherence is that participants may veer too far away from the core of the process to reap all intended benefits for it, and other participants may also encourage the

sentiment that rigor in software process is not important. If that matches the community's approach to software process, then it helps. However, if the community wants to encourage rigor, then continual disregard of process can be counterproductive.

Conversely, if rigor over relaxing criteria is always selected, it can cause community participants unsatisfied with the decision to think of software process as a hindrance to their style of work and can discourage them to take advantage of it.

5.3.7 Pattern: Pioneers don't know what all they know

Pioneers don't know what all they know

The Problem: In a team of primarily pioneers, the team members have so much shared implicit knowledge that it can be difficult to decide what needs documentation.

Therefore: It is recommended to write documentation keeping non developers or newcomer developers in mind as the consumer of that information. It may also help to go back an update old documentation, when it was found that aspects of it needed elaboration.

Resulting context and Rationale: Pioneers are often the developers drafting the initial design, requirements and rationale documentation, and a lot of subsequent documentation that build upon them. When all the concurrent consumers of the documentation are experts, a lot of common implicit knowledge can get missed. However, that knowledge may not be common or accessible to the newcomers into the community of practice.

As this kind of knowledge disparity increases, the important of pioneers to the development process in educating the less experienced developers also increases, thereby increasing the dependence of less experienced developers on the pioneers.

5.3.8 Pattern: Patron as Process Champion

Patron as Process Champion

The Problem: In a software development community, software process adoption and active practice can initially be slow and perfunctory.

Therefore: Having the Patron Role as a champion for software process helps permeate the spirit of software process adherence into the software development community.

Resulting context and Rationale: We find that if there is a software process champion, especially in an influential position like the Patron Role, it helps encourage the adoption of software process and fosters a culture of process improvement that eventually can lead to greater adoption by making software process practice the norm.

5.3.9 Pattern: Mentor as Oracle

Knowledge transfer method - Mentor as Oracle

A pattern of knowledge sharing and mentoring through employing a 'questions first approach' where the mentees lead by asking questions.

The Problem: When the mentor and mentees work together, how do the less experienced mentees arrive at the knowledge level of the mentor?

Therefore: One approach to resolve the knowledge disparity is where the mentor expects and encourages the mentees to interrupt with questions, allowing the mentor to elaborate where needed.

Resulting context and Rationale: The advantage of such an approach is that the mentor allows the questions from mentees to lead what the mentor should be explaining. The mentor does not waste time explaining things that the mentees already understand.

A risk with this approach is that many less experienced developers may not ask questions as often as they have them, they may feel embarrassed to ask questions or may feel like asking too many questions may retard the pace of work.

5.3.10 Pattern: Mentor as interrogator

Knowledge transfer method - Mentor as interrogator: A pattern of knowledge sharing and mentoring through critique using 'what if' scenarios posed by the oracle.

The Problem: When the oracle, who knows the mentees well, is reviewing and critiquing the work of experiential juniors, critique from someone so knowledgeable can appear too harsh.

Therefore: While critiquing, the oracle intentionally adopts a strategy of asking 'what if' scenario questions to allow the less experienced developers to justify their choices and consider the scenario, without feeling accused.

Resulting context and Rationale: Karoline, Casey and Natalie experienced that when working with Alex, their occasional mentor and by far, the most experienced and knowledgeable developer on the team, especially during a code review or walkthrough, Alex would pose different scenarios and questions for them and wait for them to answer to lead them to examine scenarios where the code or functionality may fail or be incomplete. This is a very effective method of ensuring that developers ask all possible questions to examine their code. The less experienced developers appreciate this, as Alex helps make their code better and more robust. However, the less experienced developers also feel put on the spot when Alex poses the different questions to them. Often, while posing the questions, Alex knows that the scenario he is asking about may not be something the less experienced developers are aware of, as it may be dependent on the kind of historical knowledge that is not written down or is not easily discoverable. Alex has mentioned that he intentionally poses the scenarios as questions instead of bluntly and directly stating that something is wrong or missing, to apply a more gentle approach than an accusatory one.

This type of strategy is best adopted when the oracle knows the mentees well and knows what to expect from their interaction together about their receptiveness to this form of disguised critique.

5.3.11 Pattern: Mentor as interlocutor

Knowledge transfer method - Mentor as interlocutor

The Problem: When a mentor or oracle, who does not know the mentees well, is reviewing and critiquing the work of experiential juniors, critique from the knowledgeable mentor can appear too harsh to the mentees.

Therefore: While critiquing, the oracle intentionally adopts a conversational tone of first agreement then offering suggestions, instead of simply pointing out mistakes or missed optimization or style coherence.

Resulting context and Rationale: Karoline and Natalie experienced in working with Leon, who is the expert on the subject matter of their work together and the most experienced developer, perhaps in the company, that when Leon has to offer critique of their code or ask them to make changes, he brings it up in a conversational tone, after some appreciation. And then, still just as part of the conversation, he casually offers a suggestion for change in the code, without using any language like mistake or wrong. As a result, the less experienced developers do not feel offended or accused. They just take his suggestion as a simple task, and not a failing on their part.

This strategy perhaps serves Leon well, as he rarely works with the other developers n person. His interaction with other developers is typically over the phone and for brief periods of time on small stories or tasks. Leon does not already have a close relationship with other less experienced developers, to employ a more aggressive approach.

5.3.12 Pattern: Encourage pair programming by just doing it

Encourage pair programming by just doing it

The Problem: Even if pair programming is the official practice, for many developers who haven't been regularly practicing true pair programming before, pairing ends up being a lot of discussing together and programming alone. How do you get a community of developers to adopt real paired programming?

Therefore: To get the team to regularly and reliably practice pair programming, if you start with a few people, especially the newcomers to start practicing paired programming, it can quickly permeate through the team and start becoming common practice.

Resulting context and Rationale: At TAI, we learned from the pioneers and early onboarders that before the newcomers Nathan, Natalie and Casey joined the software development department, what most developers called paired programming was usually discussing design or tasks together and then going away to code separately and regroup perhaps at review or integration time. This was the way the developers would pair for several years.

When Natalie and Nathan started with their onboarding conducted by Davin, as part of the onboarding, Natalie and Nathan pair programmed with Davin after a day or so of watching over his shoulder. Pair programming became the standard daily practice for Natalie, Nathan and Davin. When Natalie and Nathan joined other teams, they carried this practice with them and slowly, other more experienced developers grew accustomed to this relatively

more authentic and perhaps more effective form of pair programming.

One disadvantage of this approach is that some developers may not get an opportunity to pair with someone who now regularly pair programs due to team or product division lines. It can require conscious seeding.

5.3.13 Pattern: Communicate design rationale

Communicate design rationale Pioneers share their historical knowledge of design rationale and context for design decisions with less experienced developers.

The Problem: The real wealth of knowledge and significant advantage that pioneer developers have over other less experienced developers is the knowledge of historical design rationale and other factors that contributed to design decisions historically, or the memory of which strategies worked for the development effort. How or through what processes is this knowledge shared with less experienced developers? Some design rationale may be captured through story descriptions or through formal design rationale tracking tools, but a lot more context is remembered by pioneers than is captured.

Therefore: It is important for pioneer developers to work together with less experienced developers. A paired programming style collaboration between pioneer developers and less experienced developers from time to time would help trigger conversations where the design is questioned and the pioneer developer shares their memory of the historical context and design rationale for that particular design instance.

Resulting context and Rationale: It can be difficult to remember to capture the entire design rationale or complete context of options considered before the design choice is made. Developers remember more than they may formally record. The problem of not knowing what they know. However, when communicating with other developers, it may be easier to recall and share the design context. This is, in large part the knowledge that separates the pioneer developers from the less experienced developers. Sharing this leads to a team where all the developers have a good understanding of the design rationale over time.

5.3.14 Pattern: Initial Turbulence Sprint

Initial Turbulence Sprint

The Problem: When a development team with varying amounts of experience is practicing Scrum development together as a team for the first time and the team is also starting some long running feature sets, it can be too many new things with a team that does not know how to work together yet.

Therefore: Allow the team a sprint to figure out how to work together, and simultaneously conduct thorough requirements analysis and sound design.

Resulting context and Rationale: It can take a few sprints for the team to fine tune their process and norms for working together. Allowing for an initially busy and possibly turbulent sprint where the team learns how they work best together, at the cost of the completion of some of their targets for the sprint.

5.3.15 Pattern: Path to normalcy Sprint

Path to normalcy Sprint

The Problem: After a rocky sprint where a lot less gets done than planned, the current sprint has a lot of pending work rolled over from the last sprint, and the team has low morale.

Therefore: Allow the team to complete the remainder of stories from the previous sprint to feel a sense of accomplishment. Encourage the team to work together by utilizing the lessons they learned from the previous sprint. Let the team find its path to normal behavior.

Resulting context and Rationale: This pattern lets the team create its own rhythm of work and set of practices and pairings that work for it. This sprint allows the team to make up for the low throughput of a previous *Initial turbulence sprint*. There is still design and requirements refining for long running stories that happen in this sprint, but most activities contribute to moving towards a sustainable, normal practice.

5.3.16 Pattern: BAU Sprint

Business As Usual Sprint

The Problem: After a few sprints, when some long running feature sets are reaching completion, the work in the team seems to slow down, possibly lowering enthusiasm for work.

Therefore: Allow the team to function in the slow, sleepy mode, as long as the work is getting done. A team not interacting very much may just be choosing to interact less for a smaller need to do so.

Resulting context and Rationale: As the team reaps the benefits of planning and knowledge gathering in previous sprints, it also has less reason to actively collaborate to come to a common understanding. The pace of development may be sleepy as rounding up interesting stories with some last housekeeping type work which may not be as exciting as more busy sprints, but it helps the team practice its ability to be flexible in process, based on the need.

5.4 Pattern language

We describe here how the patterns relate to each other, forming a pattern language, where patterns precede and succeed other patterns.

5.4.1 Mentoring pattern relation

When a newcomer joins, one can either use *Pioneer Onboarding*, if the team mainly consists of pioneers. Over time, the team has a mix developers with *Pioneer Identity* and *Early Onboarder Identity*. At this stage when a developer with newcomer identity joins the team, the *Generational Onboarding* pattern can be used to sustain the community most effectively.

In terms of Buell's mentoring models, we observe that all developers at TAI, given the opportunity to mentor, choose the "Nurturing Model" [13] where they create an environment conducive to asking questions and learning through encouragement, as opposed to the "Cloning Model" of mentoring where mentors typically expect their mentees to learn by replicating what the mentors are doing.

We also observe that the same mentor may use different mentoring patterns in different scenarios. Alex employs the *Mentor as Interrogator* pattern when reviewing or critiquing code, where he asks questions to drive the mentees to examine their work more carefully or consider all scenarios, and he employs the *Mentor as Oracle* pattern when working

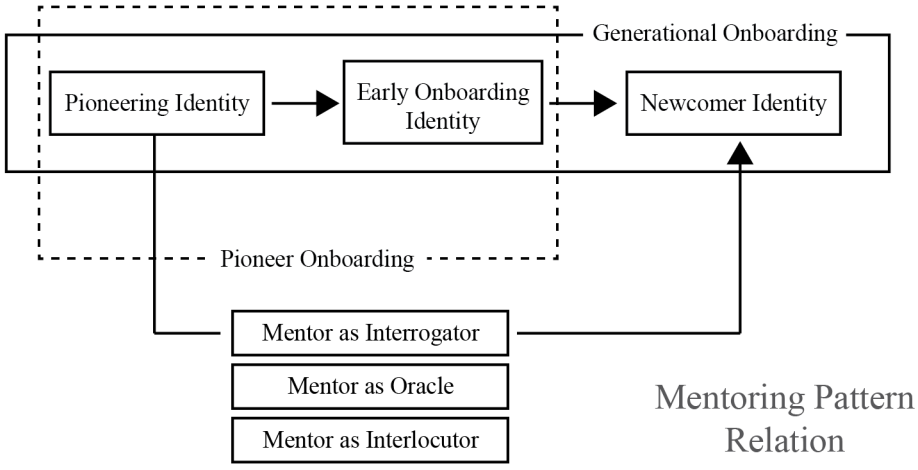

Figure 5.1: Mentoring pattern relation

alongside less experienced developers, where they ask Alex different questions and he uses their questions as a guide for what to explain and elaborate on.

In our limited experience with Leon, we found that in critique and review, he employed the *Mentor as Interlocutor* pattern. Figure 5.1 on page 134 depicts this pattern relationship.

5.4.2 Knowledge sharing pattern relation

It benefits the larger community when the development team work together to becoming a community of experts. However, it isn't always possible to only have experts, especially for growing teams. Even expert programmers may have lots to learn about the specific product or design they work on. In such a community, it becomes necessary to have mechanisms in place to share knowledge, to accommodate everyone's journey towards becoming an expert. Organizational patterns suggest an *Apprenticeship* pattern approach to move away from the *Old People Everywhere* pattern and encourage more communication and knowledge sharing between more experienced developers and less experienced developers to help less experienced developers gain the knowledge that they lack. This can eventually support a *Domain Expertise in Roles* model where the community makes or hires domain experts and then structures then structures teams around them to implement a *Subsystem by Skill* pattern approach.

We find that as *Pioneers don't know what all they know*, a good approach towards knowledge sharing is in utilizing the *Apprenticeship* pattern approach, which facilitates experts or *Pioneer Identity* developers to communicate documented and undocumented knowledge in conversation and *Communicate design rationale* with the less experienced generation of developers, through *Generational Onboarding* to prevent the *Old People Everywhere* pattern environment.

5.4.3 Process evolution pattern relation

When we see instances of process or means of working together and communicating changing at the software development community of practice at TAI, we observe the relationship of the new process related patterns we found at TAI and the existing organizational patterns that were practiced there.

To encourage pair programming in the development team, a means to allow the organic adoption of the process is described in the *Encourage pair programming by doing it* pattern where once a few developers start to adopt pair programming, subsequently who they work with are also likely to adopt the process. We witness that the existing pattern of *Self-Selecting Team* [22] facilitates the existing *Develop in Pairs* pattern allowing developers who want to work together to do so and not forcing incompatible pairings. Greater adoption of pairing also helps ensure the *Someone always makes progress* pattern, where successful pairings are more efficient and likely to make progress than individual work.

A process is more likely to be practiced and benefit the community when the developers who practice it, internalize the behavior and see immediate benefit in doing so. Creating a community where the *Developer Controls Process* pattern is effective, with support from the *Patron Role*, helps foster an environment where developers choose and adopt practices that they assess as working for them after they have tried it. We witness such a community in TAI, where the *Patron Role* supports and encourages the developers to decide and use the practices that work for them, by encouraging them to try new things, but not enforcing behavior.

Even when we see a *Process Tug of war* pattern, where the developers have a difference of opinion on adherence to prescribed process, the *Patron Role* encourages the team to determine what would work for them through dialogue and discourse.

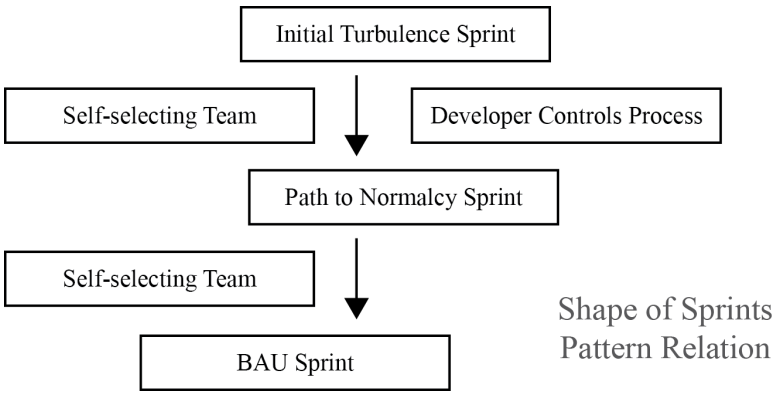

Figure 5.2: Shape of Sprints pattern relation

5.4.4 Shape of sprints pattern relation

It can take a few sprints for a mix of diverse developers to find their work rhythm and determine what practices work best for them. This distinction in needs and expectations from a sprint can influence the 'shape' of the communication in the sprint. As depicted in figure 5.2 on page 136, the team can start with an *Initial Turbulence Sprint* pattern, characterized by bursts of energy and intense communication, where the nature of the work and communication is discussed, experimented with, sometimes at the expense of the expected progress, following the patterns of *Developer controls process* and *Self-Selecting Team*. Taking advantage of a first sprint where the team decides how it works best can be followed by a *Path to normalcy* pattern where the team finishes up rolled over work and continues practicing good practices of collaboration. This can be followed by a *BAU sprint* pattern where the pace of work drops off as a lot of decision making and communication fine tuning occurred in prior sprints.

The *Yesterday's Weather* pattern [20] discusses how the velocity of one sprint can affect the velocity of the next sprint, although it is also a good predictor of the next sprint's velocity.

5.5 Relationship with research goals

After observing different types of events and patterns and trajectories over a nine month period of time, we determine how our observations serve our initial research goals.

5.5.1 Negotiation of meaning and communication

Our first research goal talks about studying the relationship between communication strategy and negotiation of meaning in the community of practice. Meaning within a community of practice can be represented by many different types of things like process, work products, artifacts, communication norms and style, forms of participation in the community of practice. Negotiation of meaning talks about the practices and processes that occur within a community, that by occurring, define what it means to be part of that community of practice. By means of participation in the community, the members define and redefine what the community does and is.

5.5.1.1 Code Review

By means of participating in a *code review*, developers define and practice the code style and standard for the community. The developers also define the norms of ownership and collaboration of the common code base. The developer who writes the code is not necessarily the sole owner of the code. The developer who reviews the code also has responsibility and ownership. As there are at often two levels of review that code undergoes and multiple levels of review that product functionality undergoes, all participants involved have the right and responsibility to ensure code and product quality. They all have the ability to question decisions or suggest improvements at any stage. Even independently writing code is not completely independent. Developers write new code for the common code base following the norms of code style and prior design decisions. So by creating new code, or modifying existing code, a developer participates in what it means to be part of community and contributes to it, adding to this existing mixed body of meaning, simultaneously accepting and redefining the 'what' and 'how' of participation and meaning - the negotiation.

When exploring and investigating any piece of legacy code, it is common practice to trace which developer wrote a suspect or ambiguous line of code and perhaps discuss with them if they remember the rationale behind them. In this way, often the person who finally committed the code to the common code base ends up leaving a trace of their partial ownership of the code, even if they were not the developer who originally wrote the code. In some way, leaving that developer's name as responsible for that line in the code log, encourages participation to further their collective understanding of the code - meaning.

The way that the code review is conducted can also vary. Code reviews for small changes while working on a sprint story is typically conducted by another developer from within the sub-team who may be familiar with the story, design and the task. When the same

code is being reviewed for a bigger code merge with another code branch, then a developer from the team or a pioneer level developer or a mix of developers of varying levels of experiences may scrutinize and review the code. As part of bug squashing, Kanban style work, it is encouraged to solicit a code review from a developer who is unfamiliar with the bug and the code, as they may be able to lend a fresh perspective on both the code and the design. Over time, it is intended by some developers to encourage the sprint developers to also seek developers unfamiliar with the code to review it. We observe that a standard and simple practice like the code review can be used to collaborate, communicate, participate and evolve membership of the community of practice, by negotiating meaning.

5.5.1.2 Mentoring

One of the most direct and deliberate means of including someone into the community of practice and through the process, describing what it means to be part of the community of practice, is mentoring. Mentoring starts from the day a developer joins and continues through their entire journey in different capacities. A developer may have multiple mentors who spend varying amounts of time with them and expose them to different aspects of participation in the community.

Some negotiation of meaning happens implicitly and indirectly, where the developers selectively absorb some knowledge and norms of practice and ignore others. They establish their own sense of what practices serve them best and what practices they choose to ignore. Over time, they introduce their own practices into the community and change the participation of other developers in the process.

Developers directly learn what is officially important to the software development community of practice and they also learn what is actually practiced. New developers are exposed to pioneers and early onboarders as mentors and learn different aspects of participation in the community which carry different levels of weight. Knowledge and customs imparted by pioneers may be more difficult to disregard, than those imparted by the less experienced early onboarders. Over time, developers get to question the value of existing practices through informal conversation and formal sprint or kanban cycle reviews and suggest changes to style or practices. Eventually, the new developers get to share their knowledge and experience with other newcomers into the software development community of practice, thereby negotiating meaning throughout their journey and trajectory in the community in different ways.

5.5.1.3 Process tug of war

Process tug of war discussed in pattern 5.3.6 and section 4.3.3.1 is an example where, while operating within a well defined process, the participants of the software development community can have disagreements on the details of process implementation or the degree of strict adherence o process suggestions. By choosing details of the process to adopt or modifying process suggestions to suit their needs, the participants define their own meaning - process.

5.5.1.4 Paired programming

As we have discussed in our pattern on the changes in the nature of paired programming over time 5.3.12 on page 130 where the process of participation is negotiated, as newcomers follow new process that permeates through to the team culture and practice. By practicing more paired programming, and encouraging others to practice it more as well, the newcomer participants in the community of practice redefine standard paired programming, as it was followed by the teams before, thereby redefining what it means to 'work together.'

5.5.1.5 Trajectory

We observe that trajectory can be influenced by communication events and choices, where a leader community member asking or encouraging another community member to participate in an event or activity or to consider a role or position, causing that member to change the path that they traverse through the community. Conversely, a community member's trajectory affects their identity, their participation and their communication acts and choices.

5.5.1.6 Open conference

We observe that by adopting a new meeting format like the open conference format for brainstorming together, breaking away into smaller teams and come together to find a nuanced resolution to a complex problem, the developers followed their leadership to expand their own practices. A thriving community of practice keeps practices evolving to suit its

needs. We find that by redefining this form of participation, the community collectively negotiates meaning.

5.5.1.7 Team deciding

As we describe in the section on team deciding work for itself 4.3.3.2, the team follows the standard definition of a self-managing team and overrides the advice of the scrum master and software department head to set its own rules for managing work. The team thereby invokes the formal definition of a self managing team and defies the common practice of taking the lead from the leaders. The team negotiates their practice and the adopted mode of team authority, in the process, negotiating what it means to participate in the team.

5.5.1.8 Minutiae disconnect

As described in section 4.3.3.5 on minutiae disconnect where more experienced developers may disagree with less experienced developers on small details of code and code style practices, depicting meaning in the form of code and style being negotiated, an example of negotiation at a small granularity. Often more experienced developers enforce their code and design preferences on the work of less experienced developers. In many of these cases, the disagreement would be in semantically equivalent details, too small to drastically change the nature of the solution.

5.5.1.9 Knowledge transfer

Through knowledge sharing practices and discussion, participants of the community of practice add to their existing body of knowledge by sharing and improving on it. This knowledge sharing can be done through different forms - formal directed knowledge sharing like presentations on new topics or sharing knowledge during formal design or process discussions or during informal conversations that are part of pairing.

5.5.2 Identity and communication

In the second research goal, we explore the relationship of communication strategy and identity.

5.5.2.1 Pattern roles

We see in the discussion of the emergent pattern roles of Pioneer Identity, Early Onboarder Identity and the Newcomer Identity, that a developer's identity heavily influences their choice of communication. The roles represent the identities of the participants and demonstrate how properties of the roles influence some communication choices, demonstrating a relationship between identity and communication.

5.5.2.2 Onboarding styles

Even within the same pioneer identity, different participants can make different choices of communication strategy, thereby defining for themselves what their role's relationship with communication choice is. We observed that pioneers often adopted a more formal, directed knowledge sharing style, whereas, early onboarders adopt a more informal and collaborative and explorative knowledge sharing style.

5.5.2.3 Code review and Minutiae disconnect

We observe that different types of onboarding styles can be related to nuances in identity and participation. We observe that two different developers with Pioneer Identity also adopt different communication strategies, as their identities have slight differences. Leon is the original pioneer who had moved away from the core product development work and was physically separated from the rest of the team and had limited experience working with this mix of early onboarders and newcomers. In light of his most recent change in identity, as an expert outsider to the main development team, and returning to regular development briefly after a few years working independently on research, Leon adopts a more conservative and conversational approach in critique. Compared to Alex, who was a pioneer who first started working under Leon and has since established himself as the foremost expert on

the primary code base and had been continuously involved with the development teams, feels like his identity is completely central to the community of practice. Therefore, he can afford to adopt a slightly harsher but more interrogatory approach when critiquing and reviewing code. Alex also feels high ownership of the code and weighs in heavily on code style decisions and design decisions, where other developers who feel less central to the community and less ownership of the code are more flexible on minor differences in style and design decisions as long as overall functionality is addressed.

5.6 Conclusion

We have fully integrated into the software development community of practice at TAI over a period of over seven months, worked in different teams using different software processes and with developers with different levels of experience. We have used interviews with different developers and other participants in the community, our participant observer's experience and an account of all communication activities that our participant observer engaged in throughout their experience, to learn how this community formed, evolved and sustained itself through change, grew in size and scope, actively and unintentionally designed its formal processes and informal practices to support a growing company.We discussed our experience and challenges in conducting this research.

We identified established Scrum and Organizational patterns that are practiced in the community and presented novel patterns that we discovered in practice at TAI. We also shared how these patterns - both previously discovered and novel, interplay and depend on each other, forming relationships and connections, consistent with a pattern language. Using the pattern format of results allows them to be easily added to the large and growing body of knowledge of the combined Organizational and Scrum patterns, and also makes them accessible to the software development community at large.

In some ways, TAI could be considered a remarkable software development community of practice. It has successfully navigated the typically treacherous waters of its inception as a start-up with different mechanical engineers producing a niche software product and its related services. TAI transformed over time by constantly improving its software development abilities as the primary product and services model of the company underwent changes. Some key aspects of TAI's success can be attributed to placing the right people in the right roles.

In spite of the presence of more experienced developers, Philip's visionary capabilities were recognized and leveraged to transform the software development effort. Philip mentions that the old, limited-communication, inadvertent waterfall way of software development

felt like a mechanical engineering approach to software development that did not fit. He worked with other experienced developers to determine what form of software development process they should adopt and after reviewing different models, choosing Agile software development with a Scrum model. Over time, the flexible, highly communicative structure of Agile, as well as market forces, allowed the software development community at TAI to grow. This growth also brought with it new and interesting challenges such as knowledge sharing, mentoring and skill management.

Possibly due to their enthusiasm for software process, the development at TAI has a variety of software process models and practices like Scrum and Kanban, even within a limited number of development teams. Using software process and in turn, thinking actively about improving development and communication practices are heavily encouraged and a culture of deliberate consideration to process is fostered.

Perhaps as a result of such a culture, the success and ongoing growth of the software development community at TAI can be attributed to strategic evolution of communication strategy in response to changing conditions and needs. There is earnest and regular discussion and evaluation of communication strategies and practices, and new and ground breaking communication practices are frequently experimented with and assessed for use. This culture also encourages the experts or pioneers to consciously practice their chosen knowledge sharing strategy, especially when mentoring less experienced developers.

We find different instances of patterns and events where we are able to perceive the relationship between negotiation of meaning, communication choice and identity. In our work with student software development, open source software development and colocated rigorous agile software development communities of practice, we find different identities of software developers based on their sense of ownership and centrality to the community. These identities are not permanent and evolve over time, forming different trajectories through and within the community of practice. A combination of identity, participation and interaction with other participants in the community of practice affects the communication and mentoring strategy.

We found that communities adopted a mix of two broad approaches in mentoring and incorporating newcomers into their community of practice. Either they share documented knowledge directly with newcomers, or they encourage newcomers to spend a lot of time with the experts and learn through interaction. We found that at TAI, they adopted the approach of encouraging learning through interaction. The amount of architecture and rationale documentation at TAI is limited and the requirements and acceptance tests are captured per story and can be difficult to understand without having a deeper knowledge of how the product works and how the architecture of the system is organized.

The primary difference between pioneers with their vast wealth of knowledge and experience and less experienced developers within the software development community of practice is that the pioneers have all that implicit knowledge of decisions past. They know the reasons why certain architectural components were connected a particular way, or why violating usability principles was necessary to satisfy domain specific customer needs or how newer libraries may offer efficient ways to design the same systems, but the software debt incurred in that effort may make it an undesirable solution.

As the software development community of practice constituted primarily of pioneers till a few years ago, a lot of the requirements and rationale has been captured by the pioneers and in some ways, *for* the pioneers. For pioneers, with their immense wealth of knowledge, when recording their findings and design intent, it can be difficult to know what is not obvious about the design, as it is implicit knowledge that most pioneers might share. Often it is only recorded what was finally done, and not what all was considered and discarded as it would be unsuitable for the system.

Developers less experienced than the pioneer developers have the ability to read the code, interpret to the extent of their knowledge previously recorded documentation and perhaps fully understand any new functionality that they built or helped develop. However, that valuable implicit historical context knowledge that pioneer developers have is something that less experienced developers only learn piecemeal over time through interaction with the pioneers.

We observe that pioneer developers, perhaps conscious of this knowledge difference may choose communication strategies that allow them to share their knowledge based on what the less experienced developers do not know, but need to. Allowing the less experienced developers to lead the knowledge sharing through their questions, which spur the pioneers to share answers focused on one thing at a time.

There are risks associated with this strategy as well. Some less experienced developers may feel like they should not be asking too many questions or may feel embarrassed to. Some developers may feel that they slow work down with their incessant question asking, especially if other developers do not ask as many questions. So in a ‘questions first’ approach to knowledge sharing, the more vocal developers benefit, and a culture of asking many questions helps encourage that.

Finding ways to intentionally encourage the sharing and dissemination of the historical, design rationale and context, and creating a culture where this form of knowledge sharing and mentoring is commonplace, may be the key to accelerating the trajectory of newcomers to pioneers, to achieve more effective communities of practice. This is a benefit of a larger culture of paying attention to and actively valuing process, which is fostered by following

the lead of an encouraging ‘Patron Role’ leader.

Chapter 6

Instruction for Software Engineering Students

In this chapter, we share our efforts and experience of integrating communication based material into the computer science and software engineering education curriculum.

6.1 Agile Communicators: Cognitive Apprenticeship to Prepare Students for Communication-Intensive Software Development [1]

We report on our efforts to enhance our undergraduate computer science and software engineering curriculum, promoting what we term *agile communication* through practice in inquiry, critique and reflection. We are targeting early courses in our curriculum, so that students internalize agile practices as part of their personal software development process. Our approach constitutes a *cognitive apprenticeship* that engages students in authentic software settings and articulates processes that are traditionally left implicit.

[1] The material contained in this chapter is a combination of material from different locations - Section 1 is a modified version of a part of the material published in the proceedings of the Agile 2015 conference under the title Agile Communicators: Cognitive Apprenticeship to Prepare Students for Communication Intensive Software Development and Section 2 is a modified version of material accepted for publication in the proceedings of the 38th International Conference on Software Engineering, 2016 under the title Instruction in Software Project Communication through Guided Inquiry

Communication-intensive activities are woven through this curriculum in a variety of ways. The POGIL framework provides a structured approach to inquiry. A program of guided inquiry through real case studies of software communication prepares students for their team software activities, and a series of reflective exercises leads them to focus on their own team communication practices.

6.1.1 Introduction: Communication in workplace and classroom

Communication — between humans — has always been a important but underappreciated aspect of software development. Although many professional software engineers are effective communicators, they typically do not have practice in articulating what it is that makes communication effective (or ineffective). That is, their knowledge remains at a tacit level, from which it is difficult to impart to students.

The situation is improving. Advocates of agile development methods in particular have always stressed the importance of flexible communication practices, deployed dynamically and strategically to maximize value. Cunningham's WikiWikiWeb, the "original wiki", contains a wealth of named patterns for agile practices, many of which fall into the domain of communication [23]. The Scrum framework is notable in this respect for the way in which it names — and therefore honors — particular communication practices that would otherwise remain tacit and invisible to students [64]. Through efforts by Sutherland, Coplein and others, Scrum rituals and principles are being captured as a pattern language of organizational techniques [22] [20].

Preparing students for the flexible, highly communicative environment of agile development is a crucial responsibility for computer science and software engineering programs in higher education. Instruction by experts in writing and communication has an essential place in the education of future software developers, but it must be matched with similar instruction within "disciplinary" courses. Computer science and software engineering instructors are in a unique position to ground the material in authentic practices, and by attending to communication in the classroom they validate it and heighten its importance in the eyes of the students.

The importance of communication in the software process is beginning to be acknowledged in the software engineering education community. The most recent version of the Software Engineering Body of Knowledge (SWEBOK) [9] has an expanded treatment of communication, with breakout sections on "reading, understanding and summarizing", "writing", "team and group communication", and "presentation skills". Recently there have been efforts to bring the expertise of writing instructors into the computer science and software

engineering curricula, engage students in authentic communication activities, and categorize the genres of communication that arise in software development setting [17].

SWEBOK now includes language that indicates that software professionals must be aware of contextual factors in their design of communication. A software developer — or a student engaged in a class software project — must think both strategically and tactically about the current problem at hand and the form of communication that will solve it most effectively. Having provided students the tools of their trade, in the form of authentic communication genres, instructors must give them guidelines for their wise use, based on the other individuals involved in the communication, the timing and location of the communication, and the form and style — in classical rhetorical terms, *audience*, *kairos* and *decorum* [36]). Students in computing disciplines enjoy problem solving and are well versed in principled approaches to solving technical problems. We want to give them similar tools for problem solving in the communication arena.

6.1.2 Goal: Agile communicators in software development

The principles of agile development, articulated memorably in the *Agile Manifesto* [27] as "individuals and interactions over processes and tools", "working software over comprehensive documentation", "customer collaboration over contract negotiation" and "responding to change over following a plan", have resonance throughout the software industry. With this shift comes a change in how we approach communication. Agile developers are also agile communicators, with the following strengths:

Proactivity: At the heart of the agile approach is a recognition that requirements, priorities and obstacles in software projects are in constant flux. Consequently, agile methods encourage patterns of constant questioning, informing and debating. Agile developers must be unafraid to inquire about requirements, to critique design choices, and to provide reflective comments on the team's process.

Flexibility: While agile frameworks such as Scrum and Kanban establish rituals and artifacts rooted in communication, these do not constitute a comprehensive, programmatic standard. Agile developers must be able to handle multimodal discourse (including written, oral and graphical communication through various media) and adapt to new communication situations, instead of relying on formal scripts and templates.

Creativity: In agile development, participants tailor the communication channels and genres they use dynamically to maximimize value, rather than cleave to a predefined plan.

Agile developers must be skilled rhetoricians, with a deep understanding of their communication options, and an ability to choose genre and style to suit the audience and purpose.

Our goal is to build these agile communication strengths in our students, through exposure to and practice in authentic software communication settings. We wish to build this exposure and practice directly into our disciplinary courses, and early in the curriculum, so that agile communication becomes a natural part of their internalized software process. Also, by recognizing the importance of communication and engaging in it at early stages, we expect to attract and retain students who are motivated by working in teams. These students are the ideal software developers of the future, since the reality is that software development is highly social and communicative.

6.1.3 A Cognitive Apprenticeship Approach: Inquiry, Critique and Reflection

The Cognitive Apprenticeship model [19] is a constructivist approach to learning that focuses on teaching concepts and practices utilized by experts to solve problems in realistic environments. It has special relevance in the context of software development, particularly in the communicative skills that we are interested in, because it emphasizes making implicit processes explicit to the learner. In typical computer science or software engineering educational settings, topics like team communication are often deemphasized in favor of more technical topics; in the workplace, the communication-related knowledge that experienced developers possess is internalized, complicating their ability to pass it along to new employees.

We have identified three fundamental agile activities that are mediated through communication: *inquiry* (strategies for resolving unknowns, coming to a shared vision, solving problems); *critique* (systematic analysis and evaluation); *reflection* (identifying and describing one's own implicit or explicit work process). Here we explain how these three components constitute a program of cognitive apprenticeship, and how we engage our students in these activities.

6.1.3.1 Inquiry

Agile development demands a spirit of constant inquiry. The famous Extreme Programming admonition to "embrace change" [5] implies continuous interaction with stakeholders

to understand ever-changing requirements, priorities and obstacles. The spirit of inquiry extends to intra-team communication; in a process of continual self-optimization, teams self-organize and solve problems together.

The basis for our inquiry based curriculum is the *POGIL (Process Oriented Guided Inquiry Learning)* approach, which originated in undergraduate chemistry education [24] and has been introduced to computing disciplines through the CS-POGIL initiative [41, 34]. At the heart of POGIL is a *guided inquiry* learning cycle of *exploration*, *concept invention* and *application*. Students work in small groups with well-defined roles — similarly to teams in agile software development — to encourage accountability and engagement. Each POGIL assignment has a common structure: supply students with initial data, guide them through leading questions that allow them to construct a unifying concept explaining the data, then provide means for them to apply and validate their newly constructed concept. It is in essence an application of the scientific method in a carefully crafted classroom setting. In addition to learning the core concepts at the heart of the assignment, students get practice in team problem solving and communication.

We have employed POGIL successfully in the third-year Team Software Project course, to introduce the concept of strategic communication in a software development setting [40]. This approach fits the topic well: the search for meaning within a given communication setting is complex, and different observers may see different patterns of communication in play. Guided inquiry allows students to take ownership of their interpretations; at the same time, we consciously steer students away from rote, simplistic answers that ignore the complexity of communication. In POGIL, students work in small groups with individual roles — a process framework similar to that of Scrum. The problem solving conversations within the groups give students further practice in team communication. Using a simple rubric based on standard rhetorical principles of audience, purpose and style, along with structural factors such as location and timing (Fig. 6.1 on page 152[2]), students characterize various communication practices, then assess those characteristics with regard to particular software project settings.

As an illustration of the exploration-invention-application cycle in practice, we give an early exercise from our Team Software Project material:

Exploration. We ask students to analyze standard Scrum communication practices (*e.g.,* daily standup) that they have been exposed to earlier, and to use our rubric as a guide to identify critical features of the communication strategies used.

Invention. From these findings, students name patterns of communication and identify contextual characteristics that make the pattern suitable for application.

[2]This figure is from the paper on Instruction by Kumar and Wallace[40]

Category	Attribute	Possible Values/ Questions
What	Scope (Input)	What elements of the project are referenced in this communication?
	Expected outcomes (Output)	What elements are subject to change as a result of the communication? What are the expected changes?
Why	Purpose	What is the purpose of the communication? Is there an official goal to be achieved? Apart from the official goal, are there other goals or needs being met by communicating?
Who	Stakeholders	Who is involved in the communication? What are the differences between participants, in terms of knowledge, needs, status, etc.?
How	Style	Is the tone of the communication formal or informal? Is there a predefined structure in place for the communication, or is the structure to be defined during the communication itself?
	Use of artifacts	Are there artifacts (tools like written material, diagrams, code) involved? Are they physical or virtual? Who owns them? Who has access to them?
Where	Location	Is there a particular place where the communication takes place? Or does it happen virtually, in no particular place? What attributes of the location are important for the communication to be effective?
When	Duration	Is there a fixed or typical duration for the communication act?
	Synchrony	Is the communication mode synchronous (with instant response) or asynchronous (with no expectation of response time)?
	Frequency	Is there an expected or common frequency for this communication act – once a project lifetime, weekly, daily?

Figure 6.1: Sample communication pattern inquiry worksheet.

Application. Next, students are asked to conjecture how the nature of a communication pattern would be affected if, one by one, different attributes of the communication act were changed: *e.g.* changing the duration of the daily standup meeting to one hour, changing its frequency to monthly, or changing it to an asychronous activity with team members "checking in" remotely. Students were asked to analyze how such changes would affect other identified attributes of the communication act such as content, perceived value, and scope.

6.1.3.2 Critique

Agile development demands continuous attention to good design, including refactoring when changing user needs and design demands dicate. Likewise, team organization and practices are also under constant review. This requires developers to be willing and able to reassess current design and practices and to articulate areas of improvement.

Traditional methods for teaching computer science — lecturing on abstract concepts, assigning a programming project related to the lectures, then grading the students' submitted finished products — resemble the outdated waterfall model of software development in many ways. An instructor writes a specification and hands it off to students as an assignment. Students toil in isolation, without the benefit of instructor feedback or team communication. When they run out of time, students submit the assignment and hope for the best – not entirely sure that they interpreted the assignment in the same way as the instructor. Lastly, the instructor applies secret tests to the student work and assigns a grade, then moves on to the next topic, regardless of whether students have successfully constructed mental models to understand the current topic.

In our Team Software Project course, students learn to critique their own work, the work of their peers, the communication excerpts from real software projects and the work products from their own software development in a way that allows them to describe the merits of different aspects identified through the critique, in relation to its use.

6.1.3.3 Reflection

A key component of agile development is continual process improvement, facilitated by periodic reflective activities. The sprint retrospective in the Scrum framework, for instance, is a ritual that encourages critique and creative problem solving. The concept of professional reflection has long been touted as a valuable means of metacognitive regulation [54], and there is evidence that it builds strong teams and projects and encourages learning [42] [59].

Once our Team Software Project students have been immersed in the Scrum cycle, iterating through multiple sprints, we ask them to reflect on their own process from a communication perspective. In the "How We Scrum" activity, they identify commonly recurring communicative acts as patterns and assess their effectiveness. Through this activity, they come to acknowledge that they have created a communication infrastructure of their own design. They then critique this design and propose improvements.

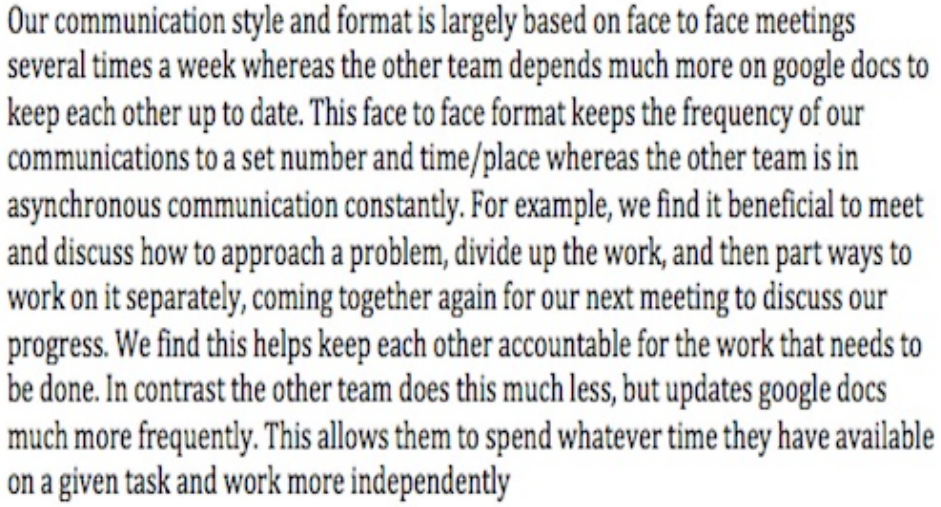

Figure 6.2: Sample student reflections: "How They Scrum" [40].

Later, in the "How They Scrum" activity, student team members interview members of other project teams about their communication practices. This contrasting perspective emphasizes the fact that they, and the other teams, have made choices that affect project performance. Sample reflective comments like those shown in Fig. 6.2 on page 154 show the influence of earlier POGIL investigations of communication strategies; in their reflections, students use the pattern approach to discuss their own communication design choices [40].

6.1.3.4 Cognitive apprenticeship

Collins [19] outlines the elements of a general framework for cognitive apprenticeship environments:

Content: Two types of content need to be taught to students: Domain Knowledge and Strategic Knowledge. *Domain knowledge* consists of the technical topics normally taught in computer science classrooms: *e.g.,* programming languages, data structures, algorithms. *Strategic Knowledge* is what experts use to make use of these classroom skills to solve real-world problems. Strategic knowledge is often difficult to express in the classroom because it is founded in experience gained from doing computer science. Here we outline the synergistic relations between Collins's areas of strategic knowledge and our approach.

Heuristic Strategies: We support student learning through the use of patterns in communication and learning [40, 15]. Software professionals routinely use sophisticated problem

solving and design skills in their communication with one another and other stakeholders in the software process. We wish to impress upon the students the importance of communication in software development, and to encourage strategic and tactical thinking about communication.

Control (Metacognitive) Strategies: Through reflection stimulated by POGIL exercises, we encourage students to learn from choices they, their teammates, and other teams make during the development process.

Learning Strategies: In our classrooms, we engage in role-modeling, role-play, and POGIL activities to help students *learn how to learn*. The cycle of doing, critiquing, reflecting, and redoing helps students develop their own learning strategies and apply them to problem solving.

Method: Our POGIL approach, with its emphasis on communication falls squarely within Cognitive Apprenticeship methods. We invite students to be *articulate* communicators, to *reflect* on their communication choices, and to *explore* new ways of approaching problems in a team setting.

Sequencing: *Increasing complexity* refers to the presentation of topics and the learning of skills in a way that builds increasingly towards expert performance. *Increasing diversity* is the sequencing of learning tasks such that a wider range of skills are increasingly required to solve problems. *Global before local* involves introducing students to high level concepts and working towards detailed implementations.

We introduce POGIL strategies in our initial computer science courses. We then introduce students to *user stories* and the notion that communication is a critical component of software development. When students reach the Team Software Project course in their third year, they are ready to learn POGIL strategies for communication and problem solving.

Sociology. Traditional teaching methods in computer science produce graduates with classroom skills and knowledge but no context or experience for applying those skills; sadly, in many cases real learning only starts when students leave academia and are faced with real-world problems. It is critical that cognitive apprentices be involved in solving real-world problems using real-world techniques as soon as possible. The Team Software Project course provides *situated learning* by having students work on authentic problems and communicate with stakeholders in the same way they will on the job. Here they engage in a *Community of Practice* [67] where they actively communicate and use the skills introduced in earlier courses. The POGIL methodology *exploits cooperation* extending learning and providing *intrinsic motivation*.

6.2 Instruction in Software Project Communication through Guided Inquiry and Reflection

In this section, we describe in detail our guided inquiry approach to addressing communication in a team software project course. This course constitutes a crucial juncture in the academic journey of our students, where they learn and practice the full responsibilities of a software engineer, including "soft skills" like communication. Early in the course, we expose the students to real communication challenges that others have faced. Later, during their project development, we ask them to reflect on the communication challenges they are facing. We describe the guided inquiry techniques that scaffold the students' understanding of communication issues, and we outline our pattern approach to communication design. We provide some initial results from the classroom, following teams as they explore the communication practices of others and reflect on their own.

6.2.1 Motivation

The inherent mutability and intangibility of software, coupled with the intense rate of change in the software development workplace, demand attentiveness and precision in oral and written communication. Students in computing-related academic programs need training from communication specialists and practice in the particular genres common to their future profession [17].

Earlier, however, we have argued that proficiency in software-related genres is not enough [38]. In most real software settings, there is no comprehensive, rote communication workflow to follow. Developers must be able to think strategically and tactically about their communication – selecting the appropriate material, location, timing and manner to suit the context. This involves analysis of the context and of candidate solutions, along with a synthesis of communication elements.

A common place for students to practice this analysis and synthesis is in a capstone project course, where they must work as a team and interact with clients. In an earlier paper [38], we compared the communication strategies employed by two different capstone teams as they engaged with their clients, their instructor, and one another. While one team forged a productive relationship with their client through effective communication, the other team failed to make progress due in large part to an ineffective listening strategy with the client. The variation in these results suggest that preparation in strategic and tactical communication, within a software development context, could help to prepare students for the

challenges inherent in project courses and in their future software careers.

In our earlier work we discussed our attempts at placing communication related material in our computer science and software engineering curricula, through weeklong trials in different junior and senior level courses. We found that students with some form of industry exposure like internships and co-ops identified and engaged with the communication patterns material, but that students without such experience need more guidance in thinking analytically about communication in the context of a software project.

The third-year Team Software Project course, a requirement for the computer science and software engineering degree programs at our institution, is an ideal venue for this kind of instruction. Building on two years of experience with programming, software design and computer systems, students take on a semester-long project, with the instructor acting as client. The technical toolset developed in introductory courses is brought to bear on a real software problem. Here is where the notion of software process – the practice of creating software products in a replicable, reliable way – can be addressed and put into action. Techniques for effective communication are obviously an important component of this agenda.

In this section, we describe the communication-focused curriculum that we have developed for the team software course. We highlight the three primary tools – guided inquiry, patterns, and reflection – that we rely on in this curriculum. We provide some preliminary data from our first course offering, in the form of survey results and excerpts from student work.

6.2.2 Background

Our Team Software Project course includes an introduction to the concept of software process, focusing on the Scrum framework [56]. One advantage of placing our instruction in this context is that Scrum explicitly acknowledges the importance of repeated, well-constructed communication [64]. Many of the iconic practices of Scrum – stand-up meetings, sprint retrospectives, planning poker – are designed to increase discussion, reflection and debate, all of which help to strengthen the software process. The message that we wish to add is that Scrum, or any other process framework, can provide only broad guidelines for communication, not narrow, comprehensive rules. For instance, team members may follow the practice of daily stand-up meetings, but it remains to their creative powers to determine what activities follow from the information shared at the standup.

We use a process of guided inquiry [12], where students construct their own interpretations of the subject matter through critical thinking and problem solving. This approach fits the

topic well: the search for meaning within a given communication setting is complex, and different observers may see different patterns of communication in play. Guided inquiry allows students to take ownership of their interpretations; at the same time, we consciously steer students away from rote, simplistic answers that ignore the complexity of communication. In the Process Oriented Guided Inquiry Learning (POGIL) [34] model that we adopt, students work in small groups with individual roles: a process framework similar to that of Scrum. The problem solving conversations within the groups give students further practice in team communication.

Another important component of our approach is the use of the pattern language concept [28] [22] for analysis and design of communication. Each communication pattern describes a set of properties associated with a communicative act. As explained in our earlier work, patterns may define particular communication genres but may also describe certain cross-cutting properties that apply to multiple genres [38]. Each communication act can be seen as an overlay of multiple pattern instances. The challenge for the students in our curriculum is to identify instances of communication patterns, assess their fit with regard to the communication context, and to consider other communication options with a better fit.

In the fall of 2013, we gave two week long iterations of our communication material in different sections of the team software project course. From this experience, we found that one week intervals, though productive, did not leave us enough time to do anything more than merely introduce the concept to students, and it did not allow students to reflect on their practices. In the spring of 2014 we introduced a course plan with communication instruction woven into the fabric of the entire term. Upon its success, we employed the same approach for a semester long course in the fall of 2014. The material described in the following section describes the instructional activities we devised.

6.2.3 Course Communication Activities

In the Team Software Project course, students complete two software projects in a semester. The first is a two-week introductory project in which students work in pairs on a common project across all pairs. The primary goal of this assignment is to give students the preparation they need for the second project: practice with the development tools they will be using, and instruction in the fundamentals of Scrum. The second project is typically performed in teams of four or five students, and students select their target application after negotiation with the instructor on feasibility and scope. Students follow three sprints, each three weeks in length. Within this course structure, we integrate several communication activities at different points in the term. We begin by guiding students through analysis of authentic software communication practices using our pattern approach. Later, when

teams have built up experience in their second project, we ask them to characterize their own communication practices, as well as those of other teams, in terms of patterns and reflect on their effectiveness.

Our primary objectives in designing these activities are to have students critique their own communication practices and the practices of others, and to use patterns of communication to identify key attributes of communication to aid their analysis. Here, we group our communication-based activities into activities analyzing project-external communication practices and activities reflecting on internal communication practices.

6.2.3.1 Analysis of project-external communication

Analyzing basic Scrum practices. At the beginning of the second project, we introduce communication analysis through a series of short guided inquiry activities. In the first session we ask students to analyze standard Scrum communication practices (daily standup, burndown chart) that they have been exposed to earlier. Students are given a short reading about these Scrum practices as homework, and in class they work together in groups to answer questions about different aspects of the communication practice they have been assigned. The initial questions ask students to characterize the communication according to the traditional "Kipling questions" of Who, What, Where, When, How and Why.

Next, students are asked to conjecture how the communication act would be affected if one by one, different attributes of the communication act were changed. For instance, changing the duration of the daily standup meeting to half a day or changing its frequency to twice in a project as opposed to being held daily. Students were asked to analyze how such changes would affect other identified attributes of the communication act such as content, perceived value, and scope. An example question from this activity is "Now imagine a scenario where the WHEN properties of a burndown chart were changed and it was updated twice in the project life cycle. How does that affect its use and relevance?"

This activity is designed to encourage students to identify different attributes of communication acts and determine how well they match the project context. We feel it is important to demonstrate that not all communication choices are equally valid; some patterns are simply ill suited for a given context. The student groups then share their answers with the rest of the class and discuss how they arrived at the answers. Analyzing student team communication. In the second guided inquiry activity, students are given excerpts from real student software project case studies and asked to identify and analyze communication acts. Our case studies come from ethnographic studies of senior capstone projects at our institution, performed as part of an earlier project [10].

In one scenario, a student brings a rough hand-drawn control flow sketch to a meeting with the client, then uses the sketch as a means for coming to understanding about the details of some legacy code. In another, a team leader sends an email message that reports results of a client meeting, includes sample code and delegates tasks to teammates. Both scenarios, presented in their authentic imperfection, invite complex critique. For instance, the hand drawn chart can be seen as an "unprofessionally" informal artifact to present to the client, but it can also be seen as an important catalyst for provoking detailed conversation between student and client. Similarly, the email message can be seen as effective in accomplishing a broad range of tasks, but packing so much disparate information into a single message can also be seen as unfocused and difficult for readers to parse. The email message used in this assignment is shown in 6.1 on page 161.

Students are first asked to identify typical concerns in the given scenarios that occur often in software projects. Students then identify attributes associated with Who, What, Where, When, How and Why, analyze them for their merits and discuss other strategies that they would have chosen. An example of the questions is "Compare the characteristics of the communication act you have identified to the typical concerns of the communication scenario that you have identified. How well does the To-Do List help address the concerns? Would you change anything? "

This activity moves the class conversation from idealized, generic descriptions of project communication to "messy" but more realistic artifacts, like the ones they will encounter and produce in their own projects. From prior experience, we know that students are primed to dismiss the work of fellow students without serious analysis. To mitigate the risk of shallow, unmeasured criticism, we include written and oral cues to guide student inquiry in a more open-minded direction. For instance, in the case of the hand-drawn control flow chart, we frame the activity within a context where the student needs to demonstrate progress after some initial delay, check with the client expert, and quickly correct errors in her understanding as they arise in discussion. These cues shift student inquiry away from an assumption that only a flawless, "gold plated" diagram is acceptable in this circumstance. Explicitly acknowledging that informal, on-the-fly material is acceptable, particularly in situations of flux in requirements and assumptions, is critical at this moment.

Analyzing distributed collaborative project communication. In the third POGIL based session, students examine email excerpts from another real case study: here, a geographically distributed open source data visualization project where all communication happens over email. Students are asked to analyze an email exchange between the host of the group list and a programmer new to the group.

Students are asked to identify (in general terms, but grounded in the given email material) the points of common understanding between the mentor and protégé and the points they

Table 6.1
Email from student project case study

The email was sent out immediately after a meeting with the client where they demonstrated their current progress and got recommendations from Hank for the next steps:
Subject: New Control code From: denise@mtu.edu To: crane-cs-l@mtu.edu Howdy, attached is the revised code for the control function as per our discussion with Hank today. There are comment blocks at the begining/end of each s-function chunk, except that part that initializes the variables for the 3 computational sections, because that still has to be divided into which S-functions need what. To do this week Bob: weekly presentation; become MatLab expert Denise: move the defines to a .h file, work out some VeryLargeVector (VLV) ideas for passing around global modifiable variables in MatLab Justin: work on splitting up the un-assigned portion of the control function into the 3 calculating s-functions. Meeting 12 Thursday - control s-function decomposition code review

are trying to resolve. Then the presence of an asymmetrical, mentor-protégé relationship is acknowledged, and students are then asked to identify features of the conversation that mitigate the risk of intimidation.

This activity exposes students to a truly authentic workplace scenario. The participants in this communication include world experts in data visualization. This underscores the message that communication choices must be made even by seasoned professionals. This case study also introduces an interesting contextual constraint: relying solely on the asynchronous, textual medium of email. Students identified different communication patterns from the excerpts and were guided to identify underlying themes like implicit mentoring between participants.

User Interviews. In the middle of the first sprint in the longer project, project teams are required to interview users, to inform the activity design [49] of their applications. For the sake of availability, students from other teams are asked to act as the users for other groups, based on characteristics of the users and the proposed applications - knowledge of music, interest in video editing, home security, cooking, etc. Each team interviews at least two mock users and then identify and analyze the user's current practices and expectations to design their application accordingly.

Students are first asked to prepare a strategy for requirement elicitation and then conduct their user interviews, without leading the users on too much to answer according to what the team wanted to hear. To guide students in their preparation, the instructor discusses the distinction between unstructured, semi-structured and structured interviews [26] [66], along with the relative advantages of each approach. The teams share the results of their interviews and give an oral report to the class on the merits of their chosen strategy and how they could improve their initial strategy.

6.2.3.2 Activities reflecting on internal communication

Once the long-term projects are underway, we ask students to reflect on their own practices at several points in their project cycle. These reflections are either done individually or as a group. In all the internal reflection activities, students are asked to also identify improvements in their communication strategy based on their reflections.

Sprint reflection. At the end of each sprint, students are asked to identify three communication patterns from the previous sprint: patterns that they found either effective or ineffective. For this activity, students are to identify and comment on specific instances of communication in their projects; for instance, a particular work-jam meeting during the

past sprint, rather than a generic commentary on work-jam activities. This allows students to identify the attributes of the communication act based on an actual event, not just on their assumptions about how it would happen.

Each team then selects three communication patterns from the collection of those submitted by individual team members. They then analyze the selected patterns for their merits and identify what worked and what did not in the use of that particular pattern. Based on these reflections, the students identify improvements and form their communication strategy and plan for the next sprint.

Individual peer evaluation. Another individual activity is the peer evaluation, where the students rate themselves and their teammates based on the previous sprint and then would reflect on what they learned about communication and teamwork based on that sprint.

How we Scrum – How they Scrum. After the projects are underway, it is useful for each team to reflect on how they have implemented Scrum, and to see how others have implemented it. This reinforces the idea that the software process and communication choices that a team makes are inevitably specific to that team, even within the confines of a particular process methodology.

In the "How we Scrum" activity, team members are asked individually to describe what their primary communication practices are and how or why they work for the team. Then team members work together to agree upon what communication practices they think define them. Special attention is given to those related to division of work, communicating progress, conveying issues and soliciting help.

In the "How they Scrum" activity, conducted towards the end of the second sprint, teams interview members of other project teams to determine what communication practices the other teams engage in and how or why that works for them. Each team interview members from at least two other teams. The team members being interviewed use their answers from the How we Scrum activity.

Standup Assessment. While the format of the daily standup meeting [56] is rather strictly choreographed, with each team member answering the "Questions Three", there is room for a great deal of variation in the amount of detail that each member provides, and the value of the information to other team members. Towards the end of the third sprint, students assess their daily standup meetings individually using a "detail vs. value" graph. Each student rates every member of the team (including him/herself) on a graph with a five point rating of "detail" on the y-axis to "value" on the x-axis. Students rate the level of detail and the perceived value of the information offered by each team member in his/her update. The individual graphs are merged to form a new graph where the placement of each member is

the average of the rating across all the individual graphs. Teams are then asked to reflect on why their average graph appears the way it does.

6.2.4 Results

While the goals of the instructional material are clear – to impress the importance of communication in software development, and to encourage strategic and tactical thinking about communication among our students – the ways in which these desired effects manifest themselves in student work is not immediately obvious. We therefore use a grounded theory approach to identify emergent themes in students' written responses [31]. In this section, we share some excerpts from student submissions that we have used to develop our coding scheme for the grounded theory analysis.

Rote responses. In the following excerpt from one of the initial activities, analyzing student project case studies, we see that the answer works within the Kipling question structure, but that it is thin in content and does not explore attributes of the activity in depth: "Who - Team and client What - Update on progress Where - Client's office When - Week 6 of development How - Not as formal/professional as it should have been."

This type of response is in keeping with our earlier weeklong interventions, where we found that students would participate actively and deeply in the class discussion, but that the answers they turned in would lack the same depth. These rote responses tend to appear in the early phases of instruction, when students are still perhaps unconvinced of the emphasis on communication, and when they do not yet have personal team experience to draw from.

In Table 6.2 on page 165, we share an excerpt of a communication pattern exercise using a template that the students were given (indicated by the first three columns – Category, Attribute, Possible Values/Questions) and the student team answers in the last column. The template, with its leading questions, does appear to provoke a broader exploration of the communication attributes:

Acknowledging importance of communication. In the first project's peer evaluation, where students individually answered what they learned about communication and teamwork from the initial two-person team project, many students reported on how they realized communication is important as the comment below illustrates: "I learned how effective communication can be in turning ideas into code. Lack of communication can be detrimental because lots of work could be re-done or removed if communication is not up to par."

Table 6.2
Sample communication pattern

Category	Attribute	Possible Values/ Questions	Student Team Answers
What	Established Scope (Input)	What is the decided purpose and scope of this type of communication act	To work on the development of the application in a group setting
	Expected Result (Output)	What is the expected outcome of this type of communication act	Implement some subset of our desired functionality
How	Style	Formal or Informal Structured or Unstructured	Informal
	Artifact type	Is there an artifact involved? What type of artifact is it? Who has access to the artifact? Who owns it?	 Code Referenced by all
When	Duration	Is there a fixed or typical duration for the communication act?	2 hours, not fixed length, actual meeting time
	Synchrony	Is the communication mode synchronous (like a face to face meeting) or asynchronous (like email)?	Synchronous
	Frequency	Is there an expected frequency for this communication act – once a project lifetime, daily?	Continually occurring meetings of varied length every meeting

Critique and redesign. After the user interview activity, some teams reported that they should have spent more time working out their strategy and that the interview would have been more productive with a strategy involving more visual cues for the subjects. They shared that for the usability testing at the end of each sprint, they would use their lessons from the user interview and have a more carefully constructed strategy.

In the following excerpt from the sprint 2 plan assignment at the beginning of the second sprint, which was one of the first reflection activities as a team, we notice that the reflection statement, while lacking in detail, does provide a springboard for analyzing previous and current practices and making a plan based on what the team learned:

"During our meetings, we had a tendency to get off track, or spend too much time trying to answer questions. To make these meetings more efficient, we will be creating a task document on Google Docs to help everyone better prepare for the meeting. This document will include any issues that anyone wants to bring up at the meeting, from questions to issues and reminders. As tasks are addressed during the meeting, we will be converting the task list into a version of meeting minutes. This can then be used as a reference for people who don't remember the results of our discussions."

On the other hand, some team responses show an ability to identify and critique a communication pattern, but not necessarily willingness or ability to change it to better suit the context: "Emails – What we do: Emails about project deadlines and meeting times are common. We don't always respond often enough, or confusion is caused by late responses. What we plan to fix: Emails will be responded to within 24 hours."

In one team's standup assessment, all team members were placed at the same value of high detail and high value. The team reflected that it was probably because they took their daily standup meeting very seriously and made sure they got a lot out of it. In one team all four members were rated low on detail, and two members were rated closely as low-value while the other two were rated closely as high-value. When asked to reflect on their graph, the team shared that the two groupings were because they had divided their team into two sets, one dealing entirely with the front end and another dealing entirely with the back end. As a result, not a lot of detail was shared in the daily standup meeting, as the two subgroups would meet frequently and did not have much to share until the end of the project.

Deeper analysis. In later communication-focused activity, we see that the nature of the teams' reflections has deepened to examine specific aspects of their communication practices. Here, we share some excerpts from the How we Scrum/How they Scrum assignment. Student groups reflected on their communication practices and the practices of other groups. This activity was conducted in the middle of the third sprint when students had participated in reflection activities of other types and had used communication patterns on

multiple occasions. We notice that the students reflect in the language of the communication pattern attributes, commenting on "style", "synchrony" and "format".

"Our communication style and format is largely based on face to face meetings several times a week whereas the other team depends much more on google docs to keep each other up to date. This face to face format keeps the frequency of our communications to a set number and time/place whereas the other team is in asynchronous communication constantly. For example, we find it beneficial to meet and discuss how to approach a problem, divide up the work, and then part ways to work on it separately, coming together again for our next meeting to discuss our progress. We find this helps keep each other accountable for the work that needs to be done. In contrast the other team does this much less, but updates google docs much more frequently. This allows them to spend whatever time they have available on a given task and work more independently."

Another team reflects on the same activity: "There are several major differences between our communication styles and the other two teams we talked to. For example, Team 2 relies very heavily on email, whereas we haven't sent an email related to the project since the very first week. We also differ in how we divide work. Team 2 divided the different portions of their project arbitrary and just ran with it, but we divided ours by experience level, which luckily also corresponds with the platforms we use on a daily basis." Here, we see an excerpt from the standup meeting analysis activity, where student teams reflect on their average "detail vs. value" graph and offer an explanation for the graph based on their team structure and communication practices: "This is likely due to the fact that only one or two of us tend to be working on major items at a time, which means that at any given meeting, they will be the only ones with anything really interesting to share. This is a remnant of the initial structure of the team, where we were each working on our own app. Now that we are all doing the same thing, we have found it hard to work concurrently, because we haven't had much time to set up the required collaboration structure. At this point, it's just easiest to continue working on the parts of the app we each know best, and hand it off when we get to the next milestone. A lot of changes tend to span multiple pieces, which just makes it even more difficult to merge changes."

Another team reflects on how their rigorous adherence to Scrum practices and strict policing by the Scrum Master led to the entire team landing on the high detail and high value end of the graph:

"Most of our team was centered around (4,4), which means that we all had a decent amount of detail and value during our stand-up meetings. We think that it came out like this because we always take our stand-up meetings very seriously. If someone is ever giving too much information, or not enough, we always make sure to notify them. We have greatly improved the quality and value of our stand-up meetings since the beginning. If we had a chart like

this at the beginning of the semester, most of our value would be very near 0. We improved a lot!"

The following are excerpts from different teams sharing their critique of their own communication practices. We note the focused nature of these critiques, referring specifically to attributes of communication patterns:

"For the Q and A sessions, the positive attribute is the synchrony of the session. The downside is the focus of the activity – as it can be lost easily. We can fix this issue by encouraging individuals to tell the team when they are sidetracked. For the Google Docs spreadsheet, the positive attribute is shared knowledge – every team member has access to the sheet, and can see tasks. The negative is the frequency – updates only really occur once per sprint, and sometimes once a week. To address this, we now plan on checking the spreadsheet daily."

"Diagrams: Positive attribute - How (artifacts): Can be referenced at any point in the future to make sure that everything is proceeding as intended. Negative attribute - Why (risk): They aren't always easily understood, except by the person creating them, and therefore must be extremely specific, which can make them messy." "Stand-up meetings: Positive attribute - What (expected result): They help to make longÂň term decisions that are difficult to make over more asynchronous mediums like IM. Negative attribute - How (style): It's difficult to enforce a formal structure in person, we easily get off-topic and it leads to a lot of time wasted that could otherwise be spent working on the project."

6.2.5 Evaluation

We wish to determine the degree to which our instruction enhances students' strategic/tactical thinking about communication, and to what extent students find this communication-focused course content authentic and useful. To assess student perceptions and feelings, we conducted an open in-class discussion and a written survey. To assess the growth in thinking about communication, we employed the grounded theory technique of emergent theme identification. While these initial evaluative efforts indicate promise in our approach, our sample size is small, and more work is needed to evaluate its effectiveness thoroughly.

6.2.5.1 Class Discussion

Towards the end of the final sprint, the students were asked about what they had learned about communication and teamwork from their previous courses and then they were asked to share some positive and negative aspects of the communication activities.

There was only one communication based freshman course that all the students in the class had taken prior to the Team Software Project course. Some of what they remembered from the course was risk management and rhetoric. A few other students had taken some other courses where they had worked in teams through some part of it. Some students shared that they had learned some things about writing documentation as part of some other courses.

When it came to sharing what worked for them about the communication activities in class, most students said it helped them be "more aware" of their practices and made them "focus on what was working" thereby leading to improvements. Many students said that they liked that the communication activities seemed to have an "industry focus" where they learned about communication in their discipline. When asked about what they would have liked to see changed about the course, some students said that they would have liked to see "more complex forms of communication", especially ones they did not have any experience with like attending a conference call and would have liked to do some role playing where they are "put on the spot" like they might be in industry and they could "practice spontaneously giving presentations more".

6.2.5.2 Survey

A survey was administered in the last week of the final sprint and consisted of 10 questions. Students were asked on a 7 point, Likert like scale whether they agree or disagree with the statement in that question. On the 7-point scale, 1 stood for Strongly Disagree and 7 stood for Strongly Agree, with 4 standing as Neutral. Out of the 30 students enrolled in class, 29 took the survey. Table 6.3 on page 170 describes the questions asked, the median, mode, standard deviation and variance and Margin of error with a confidence interval of 95

There were three main motivations behind the questions: asking about the students' understanding and ability to understand and perform the communication based activities; assessing the degree to which students considered good communication important to the software process; and assessing the degree to which the communication based activities helped the team improve their process. The responses indicate that our instructional material is effective at building awareness of the importance of communication in real software

Table 6.3
Survey questions and analysis, with margin of error for a 95% Confidence Interval

Survey Questions	Mode	Median	Margin of Error
"I understood how to perform the communication based activities"	6	5.71	0.38
"I was able to understand the rationale behind the communication based activities"	4	4.79	0.54
"I was able to use the communication patterns template easily"	6	4.79	0.59
"The communication based activities made my team more aware of our communication practices"	5	4.86	0.45
"The communication based activities helped my team improve our communication practices"	5	4.57	0.56
"What I learned about communication in this course was more relevant to my field than other courses about communication"	7	5.43	0.56
"This course has made me realize the importance of communication in the software industry"	7	5.5	0.54
"Improved communication leads to a better software making process"	7	6.32	0.30
"Improved communication leads to a better software product"	7	6.39	0.30

development (median of 7=Strongly Agree). There is also indication of some success in conveying the pattern approach as a natural way for analyzing communication – though we must take care to assess whether the perceived ease of using patterns is associated with the rote responses we encountered in some of the submitted material. The survey does indicate a need for more support in building rationale for the communication activities (median of 4=Neutral); we must take greater care to explain why we are engaging in them. Finally, the survey responses indicate a moderate level of agreement (median of 5=Somewhat Agree) that the activities helped to improve teams' communication process.

6.2.5.3 Written Assignments

In this paper we are restricting our grounded analysis to the assignments that were submitted collectively by each group. The assignments were coded for emergent themes associated with communication reflection and design. Themes that emerged include: identifying flaws in their communication strategy; identifying improvements they can make to their communication choices; critiquing communication choices of others; discussing the impact of communication on their project; and identifying the audience or style of their communication choices. Table 6.4 on page 172 lists some frequently observed codes, arranged roughly by level of sophistication.

We made multiple passes through the submitted group assignments to code them and considered each group's journey. We noticed that even though all the teams were given the same instructions and adhered to the same project timelines, their project practices differed and their reflections differed a lot. However, we did notice that the depth of reflection increased over the semester as the teams had more practice reflecting, as indicated by a greater number of coded reflections in the final project story assignment compared to Sprint plans earlier in the semester and in the quality of their written assignments.

We are particularly interested in the evolution of each team's attitudes and abilities regarding communication. At this early stage, we concentrate on individual scenarios: positive scenarios as confirmation that our approach can work, and less successful scenarios as an indication of how to improve it. As a "best case" scenario, we share the story of one team that truly seemed to embrace the spirit of reflection on communication. This team followed the structure of the Daily Stand-up meeting very faithfully and enforced the roles and responsibilities of the team members. Early in Sprint 2, the team noted that their meetings were working out quite well for them but "there's always room for improvement" and that they needed to "keep on track" better and not "spend too much time trying to answer questions" [CRITIQUE].

Table 6.4
Sample of codes used in group assignments

Code	Description
IMPACT	Students explicitly discuss the impact or importance of communication in their project
PRIOR	Students relate their current communication experiences to their prior ones
EXPLAIN	Students explain or defend their communication choices
CRITIQUE	Students offer critiques of their communication choices
ALTERNATIVES	Students identify improvements or choose alternatives to the communication practices
AUDIENCE	Students discuss the audience of the communication act
STYLE	Students discuss the style of their communication act

As a solution, the team decided to keep a shared task document to allow team members to better prepare themselves for meetings and to eventually convert the task document into a task list (instances of considering [ALTERNATIVES]). The team also reported time wasted and confusion with scheduling and reminding members of meeting times. In addition, to reliably inform members of updates to documentation and project code, the team made a policy of sending out an email when any member makes changes that might impact code that another member is working on (instances where they mention how they had to change their communication practice, carefully considering [AUDIENCE]). The team noted that even in the second sprint, "many good instances of documentation ... were overlooked ... and only seen when they were mentioned". In Sprint 2 and 3, the team reported that their user testing experience helped them learn a lot about their user's needs (instances of considering the [USER]) and make changes to their software accordingly.

The team attributed its success to its strict adherence to Scrum practices in a formal, structured manner (identifying that the [STYLE] of communication was important). In fact, in terms of complexity of delivered product, scope of project and completion of original goals, this student team would be considered the most successful. They also submitted the most reflection rich assignments (in terms of instances of reflections coded in their assignments). In their final reflection, as the Project Story assignment, they describe their earlier practices and their shortcomings, followed by the alternatives they adopted and how they panned out. On coding their group story, we found several occurrences of identifying room for improvement in their current communication strategy, and some occurrences of attention to audience and style of their communication and product.

6.2.6 Discussion and Conclusion

We found that our communication-focused curriculum produced far better results than our earlier, briefer interventions. In using this new curriculum, we also had the "home field advantage", as the instructor of the course was the person conducting the communication based activities and the activities themselves had a significant effect on the students' final grades. We believe that the deeper treatment of communication encouraged the students to consider it as something inherent and important to software engineering.

In our prior experiences, restricted to week long interventions, the students had participated enthusiastically in the in-class activities, but post-class response rates tended to be low. The students' grades were not affected by their participation in the communication based activities, which may affect how seriously students would take follow up activities and whether they would spontaneously use what they learned beyond our class time. We also found that the answers they would turn in would be very brief compared to what they would share verbally with the class. This is a trend we observed for some teams in the initial activities of the semester long intervention as well. However, towards the middle of the semester, after more experience with the communication based activities and perhaps realizing that analyzing communication is something they are expected to do all through the semester, we saw a marked difference in the quality and depth of analysis we received. By the last month of the semester, the students were comfortably analyzing communication using the attributes of the communication pattern, evaluating its merits and proposing improvements where they found the need. Even their loosely structured answers would start by describing their earlier practices, the need for improvement and their subsequent changes. We found the ability to track the student submissions through the semester allowed us to observe the change in their analytical techniques.

As instructors, we found that incorporating the communication-focused activities was easy

within the existing structure of Scrum. The sprint retrospective and planning meetings seamlessly accommodated our communication review and planning activities. We also found that incorporating several points of reflection during the project meant that students were continuously evaluating scope and feasibility of their projects alongside the communication and allowed for all projects to be managed effectively. We have shared our activities in such detail to encourage instructors to incorporate similar activities in their capstone project or other team-intensive courses. In our semester long intervention, we were able to allow the students to reflect on their own practices in many different ways. This allowed students to realize that they can reflect on their practices at different granularities – ranging from evaluating how every team member communicates during the standup meeting to assessing overall communication plans for a sprint at a glance.

We appreciate the student comments that they need more exposure to certain genres of communication that they do not encounter in other courses. We plan to explore ways to practice non-written forms of communication like conference calls. It will likely enhance students' enthusiasm for our approach if we can provide experience in authentic communication activities that are not available elsewhere.

6.3 Relationship with goals

In line with our pedagogical research goal, our work with the Team Software Project course and our continuing work in incorporating different parts of the computing curriculum with domain specific communication components are described here.

6.3.1 Build awareness

We use scaffolding and guided inquiry techniques to encourage computing students to inspect and analyze different forms and instances of real world communication that takes places in the software development industry and education. We guide students to learn inquiry and leverage that to learn critique of domain specific computing communication.

6.3.2 Incorporate skills

Building on the inquiry and critique based knowledge and skills, we train students to apply those techniques to inspect, reflect on and continuously improve their own communication practices, by examining and choosing among different communication choices. Students learn to practice and improve upon their communication strategy building techniques through periodic evaluation of their practices. Students evaluate and reflect on their teaming abilities through sprint reflection and sprint planning, they evaluate themselves and each other within the team through peer and self evaluations considering different aspects like contribution to code, ability to communicate, participation, etc. Students also learn to examine and critique other teams, by studying their practices and making comparisons with their own practices. Students also compare and evaluate different users by conducting interviews and by playing proxy users to other student teams.

Overall, students develop skills to evaluate their choices to make an educated decision, and to evaluate the consequences of their decisions to continuously improve their own practices.

Chapter 7

Summary and Future Work

7.1 Our research

Building on consensus in the software industry that communication is important to the software development process and can often determine its outcome, we established the need for understanding communication specific to software development and education. Our initial goal was to study communication practices in software development communities to learn how software development communities sustain themselves in spite of the chaos of changing personnel, requirements, roles and priorities. Our plan to accomplish the goals was to study different types of software development communities, using the framework of Wenger's Communities of Practice, focusing on the relationships between concepts like negotiation of meaning, identity, trajectory and communication strategies and practice. Notions like negotiation of meaning, identity, trajectory and reification do not exist in isolation and often manifest themselves in conjunction with each other. Our goal was also to take what we learn from software development communities and integrate it with the computing curriculum to sensitize students to their communication choices and complex communication based future roles in industry.

As we started examining different forms of software development, we noticed the recurrent theme of *mentoring*. Across all our different data sources, mentoring emerged as a predominant theme. We started with thick ethnographic accounts of student software development projects where mentoring from unexpected sources like clients and experts appeared to influence the success of student teams. As we examined online open source software development communities, informal mentoring played a role in the participants' decision to enter into and persist within a community. In our immersive participant observation of

a co-located Agile software development community, we experienced mentoring through onboarding and continually witnessed it in action between different types of developers. In the computer science classroom, especially in student project courses, we were the mentors that guided students to discover and internalize their complex future roles in the industry.

Much like the Agile development practices we were studying, our research employed a deliberately iterative approach where we adapted our strategy for different data sources. Our adaptable research strategy suited our grounded theory approach, allowing us to evolve our research focus based on emergent trends in our data and observations.

In practice, we focused on ways that software development communities welcome and integrate newcomers to help them develop their identities in software development and facilitate the trajectory of existing experienced development professionals to allow them to serve the community of practice through means commensurate with their value in the community and ways that keep them motivated by developing and evolving their identities over time. Participants with different identities reinforce and redefine what it means to be part of the community through participation.

Initial work: We started our studies of software development with student software development projects and used them as instructional material in the classroom to invoke an analytical approach to software development communication as problem solving, as student project studies would be relatable to software development students. We realized that even though we had thorough ethnographic accounts of their software development experience, the types of interactions and range of communication within student software development projects were limited and did not serve as an adequate substitute for studying the complexities of communication within and practices to sustain a software development community of practice.

Open source software development communities of practice: We chose a couple of long running online open source software development project communities, which made very different types of products. One community made software aimed at being used by developers to analyze properties of code bases. This community was geographically disparate and small, but rarely faced any serious contention. Their product serves a niche user base, where they design for other users very similar to them.

The second open source software development community makes popular software for audio editing, catering to a broader user base. Their customer base is more active, varied and substantial. The community of developers is also comparatively larger. However, we notice that in both communities, there are some core developer members and some regular developers who do not participate in much decision making and feel limited ownership in the community, and many newcomer developers whose participation is limited and sometimes

sporadic.

We observed some rich conversations in the open source software development communities of practice, as we had access to all their public forum email communication data, which allowed us to hone in on certain time periods and follow the progression of different types of conversations. However, we experienced issues establishing committed gatekeeper allies into the community, who would give us better access within the community, beyond the detailed email and code records.

After some investigation and analysis of the open source communication data, even though our communication data was rich, we realized we were missing the larger context and subtext of the community of practice. We only had an inferred knowledge of the structure and other interaction of the community. This limited our ability to analyze the community without making assumptions. We still consider the open source software development communities to have great potential for discourse analysis and a deeper investigation, provided established gatekeepers. However, for our investigative focus on mentoring, we pursued an alternative data source where we could learn about the context of the communication as well.

Co-located formal software development community of practice: We then arranged to study a small, co-located, software development community of practice, where we would be able to communicate with the participants to study the context. Our initial plan was to routinely interview selected developers from the software development community of practice. Then we got an opportunity to study the software development community as a participant observer. In this fully immersed role within the community, we were exposed to the inner dynamics and all the different forms of communication that take place. We were, however, not allowed to use any of the source code or product information or formal emails in our study, even though as a developer we had access to them. This limited our study in some capacity, but we were able to interview other developers and track our daily communication related activities and use our observations in different teams and different phases and types of software development processes.

Over almost eight months, we observed the software development community of practice and investigated their mentoring and onboarding processes, along with using interviews for tracking the trajectory of different developers and their identity and what affected it over time through different forms of reification and participation. We studied the journey of the community of practice as a whole, where it transformed over many years from a minuscule, disparate community of developers with limited interaction to a dynamic, growing, cohesive and efficient software development community that is faithful to rigorous software process and creative in evolving their practices to suit their shifting needs. We learned about the individual journeys of the participants of the community of practice.

We studied the daily communication practices of a typical newcomer developer, working our way through different teams and different types of development work. We experienced onboarding first hand and participated in different styles of software process.

We performed quantitative and qualitative analysis of the different forms of data we captured. Building on the work of the Scrum PLOP community [20] and the Organizational patterns [22] work to study patterns in scrum software development and software organization management, we chose to present our results from the software development community of practice in the form of patterns and pattern languages. This deliberate choice of pattern format of results helps make them available to members of the community of industry software developers and practitioners at large. We share our experience of conducting an ethnographic study of a software development community of practice as participant observers by publishing for the academic community of software development and engineering researchers to serve as a model of study.

Instruction in computing education: In our experience with modifying instruction in software engineering to sensitize students to their communication choices in software development and the relevance of communication in the development process, we tried several iterations of week long interventions injected into different levels of classes in computing education, till we found our 'sweet spot' for this modified curriculum.

We used a scaffolding of POGIL based activities where students are guided with inquiry, learn critique, apply their lessons in practice, and continuously improve through reflection. We initially exposed to students to standard scrum practices and guided the students to examine them using a pattern rubric. Students then examined instances of student software project communication, and investigated the nuances of it. They were then exposed to instances of software development industry communication, focusing on moments of implicit mentoring. The students practice the same form of analysis of communication to reflect on their own communication choices and strategy, as they reflect on their own agile software development sprints. They also critique the communication practices of their peers and learn how different means of communication can be used even within limited contexts.

To make this work available to the larger community of computer science and software engineering educators, we share it with the POGIL community, openly accessible to all. We are also working on actively expanding this work through our two year IUSE grant on 'Agile communicators.'

7.2 Main takeaways

A key factor in the success of the software development community at TAI is the presence of a Patron role champion for deliberate and active attention to communication and software process. The patron's enthusiasm for software process permeates the culture in the community and leads to a constant evaluation and improvement of communication practices to become the norm. This shows us that even the most unlikely candidates, like TAI which was once a small product startup with only mechanical engineers and physicists as founders, can evolve to practice a mature, rigorous, reflective and evolving software process, through conscious communication practices.

Our focus on mentoring led us to realize the importance of knowledge management, more specifically design rationale sharing. We found that the main difference between newcomers to a software development community and its resident knowledge experts - the pioneers, does not necessarily lie in skill or even experience. The disparity in knowledge is that the pioneers retain a lot of the implicit knowledge of design rationale and of reasons why certain design choices were considered but abandoned. This type of knowledge is not often well-documented and can only be transferred through conversations with the pioneers themselves. As a software development community grows, access to the pioneers becomes more expensive and also more crucial to a successful knowledge management strategy enabling communities to grow tactically, strategically nd methodologically.

We made a conscious effort to present our results on communication patterns in an addition to existing, established pattern bodies of knowledge. The libraries allow us to make our patterns available to the larger industry. Much like the newcomers into software development communities, our pattern results do not stand alone, they add to a growing body of knowledge and find their place with respect to preexisting patterns in their relationship to each other as a pattern language.

In our experience of creating and implementing disciplinary courses integrated with communication intensive elements designed to sensitize computing students to their communication choices and prepare them for their complex future roles in industry, we found that when such courses administer real world examples of communication and give students practice in critiquing and reflecting on communication choices using simple rubrics, students welcomed the material and made an appeal for more exposure to authentic instances of real world communication.

7.3 Future work

In line with our grounded theory approach to this research, one of the purposes of conducting an immersive study with a broad focus was to identify key factors that affect software development communities for further investigation. This knowledge helps us ask the question: *What are the questions?* Generatively, a broad study allows us to determine other, more narrowly focused studies to pursue.

We have identified some areas of further research to investigate. One of the main areas of further research would be to study how design rationale is communicated or passed between developer generations through the mentoring or other software process. Potts and Bruns [51] describe a possible alternate model where the reason or rationale behind design decisions are stored in a network as deliberation nodes, connected to the artifacts that elicit the explanation. A focused study on the real world application of different methods of design rationale sharing in different types of software communities would reveal and assess different, deliberate strategies in knowledge sharing.

Through our patterns, we commented on the notion of *shapes* of a sprint and how different sprints can affect each others communication shapes. It would be worth exploring if shapes of sprint communication are consistent in vastly different software communities, and determine whether our prescriptive sprint shape patterns apply to different settings.

We have studied mentoring in software development as an interactive but directed act, where mentoring happens from the mentor towards the mentee. With the recent advances in understanding of learning styles, the relationship between mentoring patterns and learning styles could be explored.

One of the main but passive players in a software development community is *code*. For our study at TAI, we were not allowed to use the source code, but for future research, it would be interesting to study the relationship between the journeys of new developers and source code metrics and style.

We studied a mature company which was almost two decades past its startup phase and had the befit of experience. It would be interesting to study how software development communities form and battle teething problems in young startups.

References

[1] Mentoring guide for small, medium, and large firms. *National Society of Professional Engineers*, 2002.

[2] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, 2011.

[3] Christopher Alexander. *The timeless way of building*. Oxford Univ. Press, New York [u.a.], 1977.

[4] John Langshaw Austin. *How to do things with words*, volume 1955. Oxford university press, 1975.

[5] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[6] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. The agile manifesto, 2001.

[7] Andrew Begel. Help, i need somebody! *Supporting the Social Side of Large Scale Software Development*, page 59, 2006.

[8] Andrew Begel and Beth Simon. Novice software developers, all over again. In *Proceedings of the Fourth international Workshop on Computing Education Research*, pages 3–14. ACM, 2008.

[9] P. Bourque and R.E. Fairley. Swebok v3.0: Guide to the software engineering body of knowledge, 2014.

[10] Ann Brady, Marika Seigel, T Vosecky, and C Wallace. Speaking of software: Case studies in software communication. *Software Engineering: Effective Teaching and Learning Approaches and Practices*, 2008.

[11] Ann Brady, Marika Seigel, Thomas Vosecky, and Charles Wallace. Addressing communication issues in software development: A case study approach. In *Software Engineering Education & Training, 2007. CSEET'07. 20th Conference on*, pages 301–308. IEEE, 2007.

[12] J. S. Bruner. The act of discovery. *Harvard Educational Review 31 (1)*, pages 21–32, 1961.

[13] Cindy Buell. Models of mentoring in communication. *Communication Education*, 53(1), 2004.

[14] Connie Bullis and Betsy Wackernagel Bach. Are mentor relationships helping organizations? an exploration of developing menteeâĂŘmentorâĂŘorganizational identifications using turning point analysis. *Communication Quarterly*, 37(3):199–213, 1989.

[15] John M Carroll and Umer Farooq. Patterns as a paradigm for theory in community-based learning. *International Journal of Computer-Supported Collaborative Learning*, 2(1):41–59, 2007.

[16] John Millar Carroll. *The Nurnberg funnel: designing minimalist instruction for practical computer skill*. MIT press Cambridge, MA, 1990.

[17] Michael Carter, Mladen Vouk, Gerald C Gannod, Janet E Burge, Paul V Anderson, and Mark E Hoffman. Communication genres: Integrating communication into the software engineering curriculum. In *Software Engineering Education and Training (CSEET), 24th IEEE-CS Conference on*, pages 21–30. IEEE, 2011.

[18] Gerry Coleman and Rory O'Connor. Using grounded theory to understand software process improvement: A study of irish software product companies. *Information and Software Technology*, 49(6):654–667, 2007.

[19] Allan Collins. *Cognitive apprenticeship: The cambridge handbook of the learning sciences, R. Keith Sawyer*. Cambridge University Press, 2006.

[20] Scrum PLOP community. Scrum plop patterns, Accessed on December 1st, 2014.

[21] Curtis R Cook, Jean C Scholtz, and James C Spohrer. Empirical studies of programmers: Fifth workshop. *Proceedings. Norwood (NJ): Ablex Publishing Corporation*, 1993.

[22] James O Coplien and Neil B Harrison. *Organizational patterns of agile software development*. Prentice-Hall, Inc., 2004.

[23] Cunningham and Cunningham Inc. Wikiwikiweb: People projects and patterns, Accessed on 7th April, 2013.

[24] Thomas Eberlein, Jack Kampmeier, Vicky Minderhout, Richard S Moog, Terry Platt, Pratibha Varma-Nelson, and Harold B White. Pedagogies of engagement in science. *Biochemistry and molecular biology education*, 36(4):262–273, 2008.

[25] National Society of Professional Engineers. Nspe's mentoring guide for small, medium, and large firms. 2002.

[26] A. Fontana and J. H. Frey. *Interviewing: The art of science*, pages 361–377. Sage Publications, Thousand Oaks, CA, 1994.

[27] M. Fowler and J. Highsmith. The agile manifesto, 2001.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[29] Gerald C Gannod, Paul V Anderson, Janet E Burge, and Andrew Begel. Is integration of communication and technical instruction across the se curriculum a viable strategy for improving the real-world communication abilities of software engineering graduates? In *Software Engineering Education and Training (CSEET), 2011 24th IEEE-CS Conference on*, pages 525–529. IEEE, 2011.

[30] James Paul Gee. *An introduction to discourse analysis: Theory and method*. Routledge, 2014.

[31] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, Chicago, 1967.

[32] Tracy Hall, Sarah Beecham, June Verner, and David Wilson. The impact of staff turnover on software projects: the importance of understanding what makes software practitioners tick. In *Proceedings of the 2008 ACM SIGMIS CPR conference on Computer personnel doctoral consortium and research*, pages 30–39. ACM, 2008.

[33] Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 10–pp. IEEE, 2005.

[34] Helen H Hu, Clifton Kussmaul, and Matthew Lang. Using pogil activities in computer science classes. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 753–753. ACM, 2013.

[35] Pamela J Kalbfleisch. Communicating in mentoring relationships: A theory for enactment. *Communication Theory*, 12(1):63–69, 2002.

[36] G. A. Kennedy. *Aristotle, on Rhetoric. A Theory of Civic Discourse*. Oxford University Press, 1991.

[37] Henrik Kniberg and Anders Ivarsson. Scaling agile @ spotify with tribes, squads, chapters & guilds. 2012.

[38] Shreya Kumar and Charles Wallace. A tale of two projects: A pattern based comparison of communication strategies in student software development. In *Frontiers in Education Conference, 2013 IEEE*, pages 1844–1850. IEEE, 2013.

[39] Shreya Kumar and Charles Wallace. Communication strategies for mentoring in software development projects. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 111–114. ACM, 2014.

[40] Shreya Kumar and Charles Wallace. Instruction in software project communication through guided inquiry and reflection. In *Frontiers in Education Conference (FIE), 2014 IEEE*, pages 1–9. IEEE, 2014.

[41] Clifton Kussmaul. Process oriented guided inquiry learning (pogil) for computer science. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 373–378. ACM, 2012.

[42] Marilyn Lamoreux. Improving agile team learning by improving team reflections [agile software development]. In *Agile Conference, 2005. Proceedings*, pages 139–144. IEEE, 2005.

[43] Jean Lave and Etienne Wenger. *Situated learning: Legitimate peripheral participation*. Cambridge university press, 1991.

[44] Jean Lave and Etienne Wenger. Legitimate peripheral participation in communities of practice. *Supporting lifelong learning*, 1:111–126, 2002.

[45] Beth L Leech. Asking questions: techniques for semistructured interviews. *Political Science & Politics*, 35(04):665–668, 2002.

[46] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[47] Richard S Moog, Frank J Creegan, David M Hanson, James N Spencer, Andrei Straumanis, Diane M Bunce, and Troy Wolfskill. *POGIL: Process oriented guided inquiry learning*, volume 2. Pearson Prentice Hall: Upper Saddle River, NJ, 2008.

[48] Peter Naur and Brian Randell. Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7-11 oct. 1968, brussels, scientific affairs division, nato. 1969.

[49] D. Norman. *The Design of Everyday Things*. Basic Books., New York, 1988.

[50] Maria Paasivaara and Casper Lassenius. Communities of practice in a large distributed agile software development organization–case ericsson. *Information and Software Technology*, 56(12):1556–1577, 2014.

[51] Colin Potts and Glenn Bruns. Recording the reasons for design decisions. In *Proceedings of the 10th international conference on Software engineering*, pages 418–427. IEEE Computer Society Press, 1988.

[52] Winston W Royce. Managing the development of large software systems. In *proceedings of IEEE WESCON*, volume 26. Los Angeles, 1970.

[53] Herbert J Rubin and Irene S Rubin. *Qualitative interviewing: The art of hearing data*. Sage, 2011.

[54] Donald A Schön. Educating the reflective practitioner: Toward a new design for teaching and learning in the professions. *San Francisco*, 1987.

[55] Ken Schwaber and Mike Beedle. Agile software development with scrum. 2001. *Upper Saddle River, NJ*, 2003.

[56] Ken Schwaber and Jeff Sutherland. The scrum guide. *Scrum. org, October*, 2011.

[57] Marcelo Serrano Zanetti. The co-evolution of socio-technical structures in sustainable software development: Lessons from the open source software communities. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 1587–1590. IEEE Press, 2012.

[58] Anselm Strauss and Juliet M Corbin. *Basics of qualitative research: Grounded theory procedures and techniques.* Sage Publications, Inc, 1990.

[59] David Talby, Orit Hazzan, Yael Dubinsky, and Arie Keren. Reflections on reflection in agile software development. In *Agile Conference, 2006*, pages 11–pp. IEEE, 2006.

[60] Sarah J Tracy. *Qualitative research methods: Collecting evidence, crafting analysis, communicating impact*. John Wiley & Sons, 2012.

[61] Michael B Twidale. Over the shoulder learning: supporting brief informal learning. *Computer Supported Cooperative Work*, 14(6):505–547, 2005.

[62] Lev S Vygotsky. *Mind in society: The development of higher psychological processes*. Harvard university press, 1980.

[63] Charles Wallace and Shreya Kumar. Communication patterns: a tool for analyzing communication in emerging computer science educational practices. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 729–729. ACM, 2013.

[64] Charles Wallace, Sriram Mohan, Douglas Troy, and Mark E Hoffman. Scrum across the cs/se curricula: a retrospective. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 5–6. ACM, 2012.

[65] Charles Wallace, Tom Vosecky, Leroy Steinbacher, Anne Mareck, Robert R Johnson, and Ann Brady. Student-based case studies in software communication. In *Software Engineering Education and Training Workshops, 2006. CSEETW'06. 19th Conference on*, pages 7–7. IEEE, 2006.

[66] S. Weller. *Structured interviewing and questionnaire construction*, pages 365–409. AltaMira Press, Walnut Creek, CA, 1998.

[67] Etienne Wenger. *Communities of practice: Learning, meaning, and identity*. Cambridge university press, 1998.

[68] Etienne Wenger, Richard Arnold McDermott, and William Snyder. *Cultivating communities of practice: A guide to managing knowledge*. Harvard Business Press, 2002.

[69] Yiqing Yu, Alexander Benlian, and Thomas Hess. An empirical study of volunteer members' perceived turnover in open source software projects. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 3396–3405. IEEE, 2012.

[70] Franz Zieris and Lutz Prechelt. On knowledge transfer skill in pair programming. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 11. ACM, 2014.