September, 2016

Sustainable Software Development through Overlapping Pair Rotation

Todd Sedano
Paul Ralph
Cécile Péraire

Sustainable Software Development through Overlapping Pair Rotation

Todd Sedano
Pivotal
3495 Deer Creak Road
Palo Alto, CA
professor@gmail.com

Paul Ralph
University of Auckland
Auckland
New Zealand
paul@paulralph.name

Cécile Péraire
Carnegie Mellon Unveristy
Silicon Valley Campus
Moffett Field, CA 94035, USA
cecile.peraire@sv.cmu.edu

ABSTRACT

Context: Conventional wisdom says that team disruptions (like team churn) should be avoided. However, we have observed software development projects that succeed despite high disruption.

Objective: The purpose of this paper is to understand how to develop software effectively, even in the face of team disruption.

Method: We followed Constructivist Grounded Theory. We conducted participant-observation of several projects at Pivotal (a software development company), and interviewed 21 software engineers, interaction designers, and product managers. The researchers iteratively sampled and analyzed the collected data until achieving theoretical saturation.

Results: This paper introduces a descriptive theory of Sustainable Software Development. The theory encompasses principles, policies, and practices aiming at removing knowledge silos and improving code quality (including discoverability and readability), hence leading to development sustainability.

Limitations: While the results are highly relevant to the observed projects at Pivotal, the outcomes may not be transferable to other software development organizations with different software development cultures.

Conclusion: The theory refines and extends our understanding of Extreme Programming by adding new principles, policies, and practices (including Overlapping Pair Rotation) and aligning them with the business goal of sustainability.

CCS Concepts

•Software and its engineering → Collaboration in software development; Software development techniques;

ESEM '16, September 08 - 09, 2016, Ciudad Real, Spain

ISBN 978-1-4503-3691-8/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2915970.2916002

Keywords

Extreme Programming, Grounded Theory, Code ownership, Sustainable software development

1. INTRODUCTION

Imagine being a software development manager when one of your top engineers, Dakota, gives notice and is moving on to a new job opportunity. You are simultaneously excited because the new position provides a great career opportunity for someone you respect, yet distressed that her departure may exacerbate your own project. How will the team overcome this disruption? Your investment in this engineer and her entire accumulated knowledge about the project is evaporating. Dakota developed some of the systems' trickiest, most important components. How long will it take the other programmers to assimilate Dakota's code? How will this affect their productivity and future development?

Conventional wisdom says that team churn is detrimental to project success and that extensive documentation is needed to mitigate this effect. Unfortunately, documentation quickly becomes out-of-date and unreliable [17], undermining this approach. During a Grounded Theory study, we observed projects succeed despite high disruption and little documentation. This raised the following research question: "How do the observed teams develop software effectively while overcoming team disruption?"

Exploring this question resulted in a descriptive theory of "Sustainable Software Development." The theory explains how a collection of synergistic principles, policies, and practices help develop software effectively while overcoming team disruption. This is done by engendering a positive attitude towards team disruption, encouraging knowledge sharing and continuity, as well as prioritizing high code quality. Here, *team disruption* refers to substantial ongoing changes in team composition, including team members joining or leaving, as well as temporary vacations or leave of absence.

In Section 2, we present related work on Extreme Programming and team disruption. In Section 3, we review how we employed Constructivist Grounded Theory to derive a descriptive theory supported by empirical data. We also present the research context, introducing both the company and one of the five projects under study. In Section 4, we describe the theory and how its principles, policies, and practices work together to achieve software development sustainability. In Section 5, we evaluate the theory using established criteria for evaluating a Grounded Theory. In the last sections, we examine threats to research validity, consider future research, and conclude the research.

2. RELATED WORK

In Extreme Programming [4], Kent Beck describes a set of interdependent practices that manage feature development (much like Scrum [24]), as well as technical practices that facilitate a collaborative team environment. Extreme Programming comprises 13 primary practices and 11 corollary practices.

One Extreme Programming practice, collective ownership, simply means that "anyone on the team can improve any part of the system at any time." Beck contrasts collective ownership with "no ownership" and "individual ownership." With collective ownership, every developer takes responsibility for the whole of the system. When developers see opportunities to improve the code, they go ahead and improve it if it makes their life easier [3]. Later, "collective ownership" was renamed to "shared code" [4].

One Extreme Programming practice contributing to collective code ownership is pair programming. Pair programming is where production code is created by two developers working together at a single computer [4]. Extreme Programming does not prescribe how pairs are formed or for how long a pair works together. Williams presents a pair rotation strategy for maintaining specialization, by "choosing the right partner for the right situation" [30]. People are assigned modules of the code to own based upon expertise and find partners from neighboring modules. While knowledge is shared with pairing, one person owns every story that touches their part of the system, building individual code ownership. In one case study, the project started with a pair rotation strategy based on skillsets but evolved into a daily rotation determined randomly [28]. Some teams use a pair programing matrix [1] (also called a pairing ladder [9]) to track who has paired with whom for the purpose of pairing people who have not paired recently.

Truck Number is "The size of the smallest set of people in a project such that, if all of them got hit by a truck, the project would be in trouble." [19]. Truck Number, or Bus Count, reminds management about the effects of disruptive events for a team. In 1994, Coplien [8] mentions "Truck Number" as a risk to his Solo Virtuoso pattern of using only one talented developer to create a software system. Awati suggests that Truck Number can be increased by reducing complexity, cross-training, and documentation [2], all of which are found in Extreme Programming. However, Extreme Programming implements "documentation" as discoverable, intention revealing code. Ricca [22] examines the difficulty in computing the Truck Number.

Rigby quantifies turnover using knowledge at risk analysis on abandoned files [23]. A line of code is abandoned if the most recent contributor has left the company. A file is abandoned if more than 90% of the lines in the file are abandoned. Izquierdo examines how teams managed orphaned code [14]. Joseph examines job turnovers to understand the reasons developers leave their company [15].

3. RESEARCH METHOD

3.1 Constructivist Grounded Theory

We adopted Grounded Theory [6], which provides an iterative approach to data collection, data coding, and analysis resulting in an emergent theory. We selected Charmaz' Constructivist approach to Grounded Theory, which "emphasizes understanding and acknowledges that data, interpretations, and resulting theory depend on the researcher's view" [26]. The two primary data sources were field notes collected during continuous participant observations of a seven-month project and interviews with Pivotal software engineers, interaction designers, and product managers. Interviews were recorded, transcribed, coded, and analyzed using constant comparison. In addition, the first author was involved in four other projects as participant-observer.

Grounded Theory immerses the researcher within the context of the research subject from the point of view of the participants. As the research progresses, Grounded Theory allows the researcher to "incrementally direct the data collection and theoretical ideas." The theory provides a starting place for inquiry, not a specific goal known at the beginning of the research. As we interact with the data, the data influence how we progress and alter the research direction. When starting a Grounded Theory research study, the core question is, "What is happening here?" [11]. Our initial core question was "What is happening at Pivotal when it comes to software development?"

3.2 Participants

The first author interviewed 21 interaction designers, product managers, and software engineers who had experience with Pivotal's software development process. They were distributed across four different Pivotal offices. Interaction designers identify user needs predominately through user interviews; create and validate user experience with mockups; determine the visual design of a product; and support engineering during implementation. Product managers are responsible for identifying and prioritizing features, converting features into stories, prioritizing stories in a backlog, and communicating the stories to the engineers. Software engineers implement the solution. Participants were not paid for their time.

3.3 Data Collection

We relied on "intensive interviews," which are "open-ended yet directed, shaped yet emergent, and paced yet unrestricted" [6]. Open-ended questions were used to enter into the participant's personal perspective within the context of the research question. The interviewer attempts to abandon assumptions to better understand and explore the interviewee's perspective. Charmaz [6] contrasts intensive interviews from informational interviews, which facilitate collecting accurate 'facts' and investigative interviews that attempt to reveal hidden intentions or expose practices and policies.

The initial interviews were open-ended explorations starting with the question, "Please draw on this sheet of paper your view of Pivotal's software development process." The interviewer specifically did not force initial topics and merely followed the path of the interviewee.

While exploring new emergent core categories, whenever possible, subsequent interviews were initiated with open-ended questions. For example, asking the participant, "Please draw your feelings about the code" often resulted in conversations about code ownership.

Each interview was transcribed into a Word document with timecode stamps for each segment.

In addition to collecting data from interviews, the first author collected field notes while working as an engineer on the project described in Section 3.6. The field notes comprised multiple paragraph entries recorded several times a week,

collected over seven months. These notes described individual and collective actions, captured what participants defined as interesting or problematic, and included anecdotes and observations.

3.4 Data Analysis

Data analysis began with line-by-line coding as recommended by Charmaz [6]. Coding line-by-line helps the researcher identify nuanced interactions in the data and avoid jumping to conclusions. The data then advanced from these initial codes to focused codes, focused codes to core categories, and core categories to an emergent theory.

We reviewed the initial codes while reading the transcripts and listening to the audio recordings. We discussed the coding during weekly research collaboration meetings. To avoid missing insights from these discussions [11], we recorded and transcribed them into grounded theory memos.

As data was collected and coded, we recorded initial codes in a spreadsheet and we used constant comparison to generate focused codes. Only ideas expressed by multiple interviewees informed focused codes and subsequent analysis.

We constantly compared new codes to existing codes to refine codes and eventually generate categories. We periodically audited each category for cohesion by comparing its codes. When this became complex, the codes were printed on index cards, and then arranged and re-arranged until cohesive categories emerged. We wrote memos to capture the analysis of codes, examinations of theoretical plausibility, and insights.

Constant comparison allowed us to identify "the conceptual relationship between categories and their properties as they emerged" [12], leading to a resulting descriptive theory. The resulting theory is presented in Section 4 and illustrated in Table 2. The table includes the main categories and their organization into principles, policies, and practices. Examples of quotes leading to some categories are presented in Table 1.

3.5 Research Context

Pivotal is a large American software company (with 16 offices around the world), which provides solutions for cloud-based computing, big data, and agile development.

This study focuses on one Pivotal subsidiary, Pivotal Labs, which provides agile developers, product managers, and interaction designers to other firms. Its mission is to deliver highly-crafted software products and provide a transformative experience for clients' engineering cultures. To change the client's development process, Pivotal combines the client's software engineers with Pivotal's engineers at a Pivotal office where they can experience Extreme Programming in an environment conducive to agile development. For startups, Pivotal engineers might be the first to work on the project. For enterprise clients, Pivotal provides additional engineering resources to accomplish new business goals.

Pair programing ability is a strong pre-requisite for becoming a Pivotal Labs developer. During job interviews, applicants engage in multiple pair-programming rounds to reveal their ability to listen to and empathize with pairs.

Typical teams include six developers, one interaction designer, and a product manager. The largest project in the history of the Palo Alto office had 28 developers while the smallest had two. Larger projects are organized into smaller coordinating teams with one product manager per team and one or two interaction designers per team.

Table 1: Quotes for Selected Categories

Engendering Positive Attitudes Toward Team Disruption
"I'm excited when a new person joins the team. That person has experience that might add something to the project."
"I like that people bring new energy. Projects often get into the state of a lull with the same people working on it and have the same cadence. New people bring a new perspective. [Two engineers recently joined] and it was really cool to see their fresh perspective. I always like people joining a project."
"Team members go out of their way to make new teammates feel welcome and help ramp them up."
Team Code Ownership
"I feel ownership of the code as a whole, and I feel empowered and able to go and work on any part of the codebase."
"I don't feel like I have [individual] ownership. It's really a collaborative effort to achieve where we are today . . . I feel like everybody owns this product."
"There is a lot of emphasis that you are not your code."
"I never feel like a specific piece is mine or something belongs to other people."
Overlapping Pair Rotation
"To make sure that knowledge silos don't form we rotate pairs. As people work on specific stories and specific parts of the code, we want to share that knowledge."
"Rotating pairs reduces knowledge silos and reduces the bus factor. We do not want the departure of one developer from the project to cripple the project."
"We rotate pairs because everyone has a different set of knowledge. When you work with someone you get a little bit of that knowledge. The more you pair with them, the more knowledge you get."

Commonly utilized technologies include Angular, Android, backbone, iOS, Java, Rails, React, and Spring. These are often deployed onto Pivotal's Cloud Foundry.

Pivotal Labs has followed Extreme Programming [4] since the late 1990's. While each team autonomously decides what is best for each project, the company culture strongly suggests following all of the core practices of Extreme Programming, including pair programming, test-driven development, weekly retrospectives, daily stand-ups, a prioritized backlog, and team code ownership.

We only observed teams at Pivotal Labs. Other teams, especially teams in other divisions, might have a different culture and follow different software practices.

3.6 Project Context: Project Quattuor

While we observed five projects in total, this paper focuses on Project Quattuor. This project shares many similarities with the other four. To preserve client confidentiality, we can

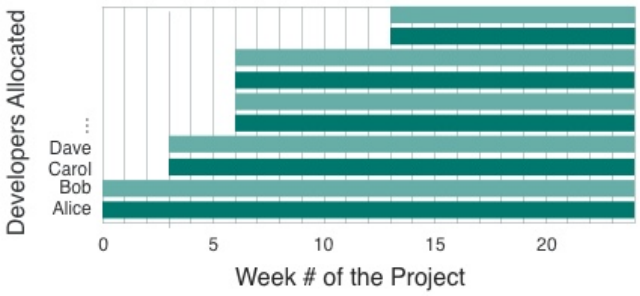

Figure 1: Planned Developer Staffing

only reveal that Project Quattuor's purpose was to develop a mobile application for controlling expensive equipment.

The project lasted 43 weeks. The initial four weeks, called Discovery and Framing, include four main activities: 1) interaction designers investigate user needs through user interviews, 2) product managers define the features for the initial release based on those needs, 3) interaction designers create an initial interaction design and validate their mockups with users, and 4) engineers mitigate technology risks. Discovery and Framing was followed by code implementation, resulting in two releases to both the Apple store and Google Play store.

The 35-person project consisted of an iOS team of ten engineers, an android team of ten engineers, and a Java back-end team of eight engineers with the support of two to four interaction designers and three product managers. Here we focus on the iOS team. The first iOS release to the Apple store occurred in week 23. Given the success of the project, the client extended the engagement for a second iOS release that happened on week 43.

Figure 1 shows the staffing plan at the start of Project Quattuor. The plan was to start the project with two developers, while adding more developers as more tracks of work became available. Figure 2 shows the actual staffing, which is quite different from the plan.

The bar chart on the top of Figure 2 shows when individual developers started and stopped working on the project. Five developers were on the project for most of its duration, while 22 people worked on the project in total. The maximum team size was 12 developers working together at the same time. The graph on the bottom of Figure 2 shows the total number of developers allocated to the project at any given week. Developers ramped up from week 5 to week 12, with an average team size of 10 and a maximum of 12 developers.

Developers were routinely rotated and were replaced for various reasons, including promotions, medical leave, leaving the company, transferring to a different office, and vacations. Atypically, the client was more concerned with feature development than cost, so absent developers were replaced, leading to 22 different people working on the same ten-person project.

The ongoing rotation of team members likely undermined the team's sense of identity [27]. In addition, the project experienced many challenges, including not having access to production back-end systems or expensive dependent physical components, and cultural differences between Pivotal and the client's deployment organization. Yet the team successfully completed the project. The client was delighted, even claiming that the team delivered a multi-year project in five months by delivering the first release.

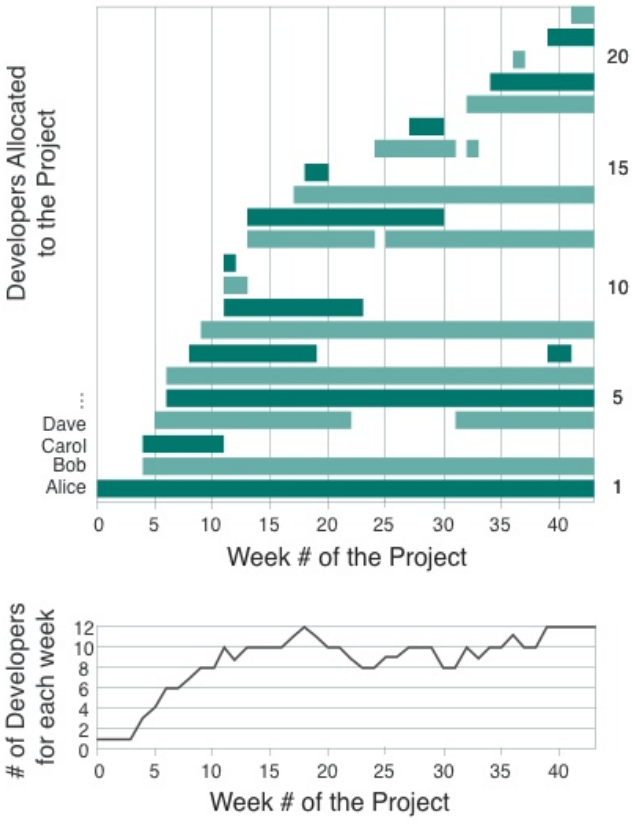

Figure 2: Actual Developer Staffing

Contrary to conventional wisdom, high team disruption did not appear to negatively influence the success of Project Quattuor. This observation raises our research question: "How do the observed teams develop software effectively while overcoming team disruption?"

4. THEORY OF SUSTAINABLE SOFTWARE DEVELOPMENT

Sustainable software development refers to the ability and propensity of a software development team to mitigate the negative effects of major disruptions, especially team churn, on its productivity and effectiveness. Our theory of Sustainable Software Development, summarized in Table 2, is targeted towards software developers and has emerged from the Grounded Theory research described above. We hypothesize that sustainability emerges from synergistic principles, policies, and practices, which collectively explain how the observed Pivotal teams overcome disruption. The ability of any pair to work on any story while caring about the code is the primary mechanism by which these principles, policies, and practices mitigate disruption.

In this section, we document each principle, policy, and practice. For each policy and practice, we present how it is used at Pivotal, and discuss anti-patterns and potential

Table 2: Theory of Sustainable Software Development: Principles, Policies, and Practices

Sustainable Software Development			
Underlying Principles	**Policies**	**Removing Knowledge Silos Practices**	**Caretaking the Code Practices**
Engendering Positive Attitudes Toward Team Disruption	Team Code Ownership	Continuous Pair Programming	TDD / BDD
Encouraging Knowledge Sharing and Continuity	Shared Schedule	Overlapping Pair Rotation	Continuous Refactoring
Caring about Code Quality	Avoid Technical Debt	Knowledge Pollination	Supported by Live on Master

alternatives. We provide deeper descriptions for practices rarely documented in the literature.

4.1 Principles

4.1.1 Engendering Positive Attitudes Toward Disruption

Conventional wisdom says that team disruption should be avoided. Yet, team disruption is a reality in the industry, as exemplified by Project Quattuor where only five of 22 developers worked on the project for most of its duration (see Figure 2). However, the observed organization engendered a positive attitude towards disruption, transforming a challenge into an opportunity and hence demonstrating remarkable business agility. Team members rolling off the project were replaced as needed. New members rolling onto the project were viewed as an opportunity to improve the current code base by providing a fresh perspective. When a new team member did not understand the code base, he or she revealed issues with code discoverability. New team members often questioned the team's assumptions and challenged "cargo culting."

The first underlying principle of Sustainable Software Development is engendering an open and positive attitudes towards team disruption, transforming a challenge into an opportunity to improve code quality.

4.1.2 Encouraging Knowledge Sharing and Continuity

Despite the fresh perspectives added by new team members, team disruption can precipitate in significant knowledge loss for the organization. Policies and practices that encourage knowledge sharing and continuity mitigate this risk. These policies are Team Code Ownership, and Shared Schedule, while the practices are Continuous Pair Programming, Overlapping Pair Rotation, and Knowledge Pollination (which are discussed below).

The second underlying principle of Sustainable Software Development is encouraging knowledge sharing and continuity, enabling the knowledge to spread from one developer to the next, and eventually reach the entire team. Knowledge sharing and continuity make the team more resistant to disruption.

4.1.3 Caring about Code Quality

Enabling knowledge sharing and continuity does not guarantee sustainable development if the team starts incurring technical debt [18]. A set of policy and practices aimed at taking good care of the code itself mitigates this risk. The policy is Avoid Technical Debt, while the practices are Test-Driven Development / Behavior-Driven Development and Continuous Refactoring (which are discussed below).

The third underlying principle of Sustainable Software Development is caring about code quality, hence avoiding technical debt and enabling sustainable team productivity.

4.2 Policies

4.2.1 Team Code Ownership

Description: Team code ownership is the extent to which any team member can modify any part of the team's code. Code ownership is influenced not only by official policy but also each developer's familiarity with and emotional relationship to the code.

Purpose: Everyone on the team is responsible for the team's code. Simply saying "Any team member can modify any piece of the code" is not sufficient to achieve the desired result of team code ownership. We documented five factors that affect the team's sense of code ownership and eight risks observed on Pivotal teams [25]. Achieving team code ownership requires a set of enabling practices. These enabling practices aim at removing knowledge silos and taking good care of the code, as described in the following sections.

At Pivotal: Every developer is empowered to work on any part of the team's code and is encouraged to refactor any code section to improve its quality as needed, especially in cases of low code discoverability and readability.

Anti-pattern: Removing team code ownership makes sustainable software development challenging. Every line of code written via strong ownership might create a knowledge silo. Code reviews are a mitigation strategy with an asynchronous delay. When the delay is too long, merging code onto the master becomes problematic, which discourages Continuous Refactoring.

4.2.2 Shared Schedule

Description: Shared Schedule signifies that all team members have the same work schedule.

Purpose: Shared Schedule enables Continuous Pair Programming, Overlapping Pair Rotation, and Knowledge Pollination practices. With Shared Schedule, teams form new pairs at the beginning of the day. The evening becomes a natural interruption to the continuous software development workflow.

At Pivotal: Team members at the Palo Alto office work Monday to Friday from 9:00 am to 6:00 pm. This is done without management coercion; each team member agreed to this fixed schedule to achieve the benefits of Sustainable

Software Development. While Shared Schedule is the norm, exceptions are possible.

Pivotal prefers co-located teams in order to promote synchronous and osmotic communication. Project Quattuor was an exception with the team split between Palo Alto and San Francisco. Each day, developers in one location remotely paired with developers in the other location to spread the knowledge across the two offices.

Anti-pattern: Flexible work hours potentially jeopardizes Continuous Pair Programming, Overlapping Pair Rotation, and Knowledge Pollination practices. A team with flexible work hours might find it difficult to pair program on all stories (as described in the Continuous Pair Programming practice). A team member consistently soloing from 8:00 am to 10:00 am might be building knowledge silos.

When developers arrive whenever they feel like it, rotating pairs (as described under the Overlapping Pair Rotation practice) becomes awkward, as there is no longer a natural time to rotate pairs. Trying to schedule a time midday to rotate pairs feels artificial. Even if the team says they will rotate later in the day, once pairs get into their stories and form context on what needs to be done, they typically forget about re-pairing until it is time to go home.

Pivotal experimented with pairing when developers arrived, but this meant that developers coming early were making decisions for the team members who arrived later, hence loosing some benefits of pair programming.

Alternatives: A possible mitigation strategy could be to adopt core work hours. Individuals would solo on simple cleanup chores outside of core hours, and switch to pair programming for feature development when the whole team is in the office.

4.2.3 Avoid Technical Debt

Description: Technical Debt refers to delaying needed technical work, by taking technical shortcuts, usually in pursuit of calendar-driven software schedules [18].

Purpose: Avoid Technical Debt enables a team to balance feature development with Continuous Refactoring (as described under the Continuous Refactoring practice). When a team is pressured to finish work by a deadline, they might be tempted to focus on feature delivery, take on technical debt, and stop refactoring. When a team delays refactoring and takes on technical debt, the code becomes harder to work with, which in turn makes it more difficult for developers to rotate onto that part of the code base. There is a dialectic tension [21] between Continuous Refactoring and delivering more features while accruing technical debt.

At Pivotal: A pair tends to create well-crafted code by avoiding shortcuts and short-term fixes. The team codes for the "present" by building the simplest solution for the current story. The team eschews over-engineering for potential future features. The team avoids technical debt by building the best solution for the moment at hand. When inheriting a large code base with existing technical debt, we observed a team actively paying down technical debt while delivering new features.

Anti-pattern: On Project Quattuor, the product manager suggested that the team deliver more stories at the cost of technical debt to make a release date. Some team members followed this suggestion, skipped the refactoring step, and introduced harder to maintain code. This decision made it difficult for pairs to rotate onto parts of the code. Pairs making the decision to skip refactoring caused future pain for the next pair to work with that part of the code. Immediately after the first release, the team spent several weeks refactoring the code to pay down the debt and consistently deliver new features again.

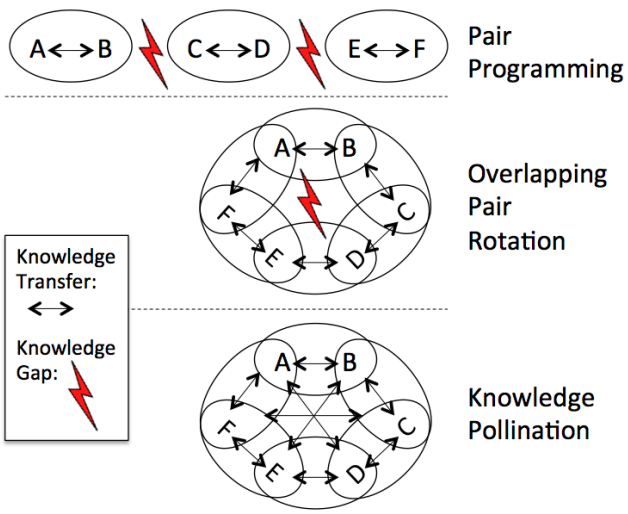

Figure 3: Three Levels of Knowledge Sharing

4.3 Removing Knowledge Silos Practices

This section presents practices for encouraging knowledge sharing and continuity, enabling the knowledge to spread from one developer to the next, and eventually reach the entire team. This phenomenon is illustrated in Figure 3, where letters A to F represent six developers working in pairs.

4.3.1 Continuous Pair Programming

Description: Continuous Pair Programming is two developers collaborating to write software together as their normal mode of software development.

Purpose: When two developers work together, they are likely to bring more knowledge, and generate more diverse solutions compared to a solo developer. Additionally, there are many documented benefits of pair programming [30]. When two developers work together, knowledge spreads from one developer to the next [31], as illustrated in Figure 3. Overall, pairing reduces knowledge silos and can improve code quality.

At Pivotal: Pairing happens with two monitors, two keyboards, two mice, and one computer. Developers always work in pairs, unless exceptional circumstances arise. For instance, solo programming occurs when one developer is out of the office for part of the day (e.g. at the doctor's office), out of the office the whole day (e.g. out sick), or involved in another business activity for a few hours (e.g. interviewing candidates, scoping a new project). When solo programming, developers take low-risk chores, refactorings, or stories. With any sizable project, there usually is something the team has been meaning to do that one person can safely do and report back to the team on its completion.

Anti-pattern: Removing this practice results in solo programming where there is a clear owner for the code written.

This would increase individual ownership and start creating knowledge silos.

Alternatives: In solo programming, to remove silos, developers could take the stories for the part of the code they know least about. Assigning stories to developers who have the least understanding of the code could be a hard sell to management as it reduces productivity (at least initially). Bird [5] suggests that this approach would introduce more defects.

4.3.2 Overlapping Pair Rotation

Description: Overlapping Pair Rotation happens when there is a rotation of the people working on a track of work: one developer rolls off the track and another developer rolls on, keeping continuity of one developer at each rotation. This results in knowledge continuity for a track of work, as illustrated in Figure 3. Typically, rotations happen in the morning as the evenings provide a natural interruption to the work.

Purpose: The rotation of developers helps spread knowledge and promotes team code ownership. The goal is to prevent the situation where one or two developers understands how part of the system works and must be assigned any story related to that part of the system. The entire team should be able to modify the code. Rotation helps prevent knowledge silos and individual code ownership from forming.

At Pivotal: Whenever a knowledge silo begins to emerge, the team actively fights against it and tries to spread that knowledge around through pair rotation. During the study, three strategies were observed.

Optimizing for people rotation: Most teams rotate based on who has paired with whom. Developers try to pair with the person they "least recently paired with" (basically a Least Recently Used strategy). Some teams use rotation techniques or tools to track this information.

This strategy does not clearly articulate the purpose of knowledge silo removal and the need for knowledge transfer. As an example, developers who recently left a track of work might ask to be rotated back without realizing the potential cost to the team. This prevents an opportunity to spread the knowledge to the rest of the team. (This issue is more serious on larger teams; on a four person team, this is not an issue).

Optimizing for personal preferences: A few teams allow developers to pick with whom they will work or on which stories to work based on individual preferences. This has the same downsides as the previous strategy.

Optimizing for context sharing: A few teams are experimenting with rotating onto a track the person who has not been working on the track for the longest time. The goal each day is for the developer leaving the track next to empower the developer who will remain on the track. Before any rotation, the remaining developer is asked, "Was enough context shared with you?" If the answer is no, then the first developer does not leave and the pair continues to work together for another day. This provides a feedback loop on how well the team is transferring knowledge.

Anti-pattern: Removing this practice means that developers can work on the same part of the code base for extended periods of time, developing individual code ownership and knowledge silos. One participant described their experience at a previous company that follows Extreme Programming. Developers could be paired for more than a month working on only one part of the system. This lack of pair rotation led to deep knowledge silos.

Ideally, developers work on the next, non-blocked story at the top of the backlog. When developers start skipping down the backlog, it can be an indication that they might not have enough context to work on any story. On Project Quattuor, a knowledge silo emerged around a complicated bug related to an obsolete technology that only a handful of people understood. Often developers would skip over stories and bugs related to that technology. At one point, the product manager reminded the team to keep "working from the top of backlog."

Sometimes a developer wants to see a story through to completion over multiple days. Maybe he or she enjoys the technology or the feature. In these situations, agreeing to the request may result in forming knowledge silos and creating a sense of personal ownership. Statements like "We need Marion on that story, only she really knows the Apple watch code base," or "Shea knows the ins-and-outs of the legacy integration, we need him to work on this story," suggest that knowledge silos have emerged.

Alternatives: Team members that build a knowledge silo can share what they learned through a demo, code walk through, or a team huddle. This helps a team share knowledge, but is less effective than working directly with the code.

4.3.3 Knowledge Pollination

Description: Knowledge Pollination refers to the set of activities contributing to knowledge sharing in an unstructured way. Examples include daily stand-up meetings, weekly retrospections, writing or sketching on whiteboards, overhearing a conversation, using the backlog to communicate current status about a story, calling out an update to the entire team, or simply reaching out to others to ask questions as needed.

Purpose: Knowledge Pollination contributes to spreading knowledge among the team as illustrated in Figure 3.

At Pivotal: Daily standups create awareness of who is working on what. Teams can write down a "parking lot" of issues to discuss during daily standups. A pair may record the current status of a blocked story so that the next pair picking it up knows the situation. Osmotic communication helps when a developer overhears another pair discussing an issue and offers needed knowledge. Instead of thrashing, a pair interrupts another pair to gain the needed information. Thus, interruptions are encouraged because they make the entire team more efficient as knowledge pollinates across the team.

Calling out an update to the entire team might be a simple as shouting "The build is broken, we are looking into it", or this interchange: "We just checked in a presenter," followed by "We just used your presenter. That's great collaboration."

While working on a story, a pair may discover that they are missing some key context that prevents them from efficiently proceeding. If the issue is about the acceptance criteria for a story, they clarify with the product manager. If the issue is about the code base, the pair can ask the people who recently worked on that section of code, or ask the entire team. To determine whom to ask, the pair may remember who did what at stand-up, look through Pivotal Tracker (an agile project management tool) to see who worked on

a story, or check out source code version history (e.g. git annotate). Two-, four-, and six- person teams seem to have collective memory of who worked on which features from daily standup.

These mechanisms help a team build awareness. Chong observed that "transmission of awareness information is a relatively effortless act in the XP environment" in her ethnographic study comparing an Extreme Programming team to a traditional team [7].

Anti-pattern: An organization that provides little opportunity to share knowledge leads to wasted time as developers must acquire the knowledge through other means or end up reinventing the wheel.

4.4 Caretaking the Code Practices

4.4.1 Test-Driven Development, Behavior-Driven Development

Description: In Test-Driven Development (TDD) developers write unit tests before creating a design and writing code. In Behavior-Driven Development (BDD) developers implement acceptance tests before creating a design and writing code. Most lines of production code are tested before the production code is written. The software's design emerges from the tests and subsequent refactorings.

In Extreme Programming, Kent Beck describes his corresponding "Testing" practice as developers writing "automated unit tests" and implementing customer provided "functional tests" for story acceptance [3]. Later, he refines these ideas as "Test-first programming" [4].

Purpose: This practice creates a safety net and empowers a pair to have the confidence to modify the code base. This enables any pair to pick up any story. Continuous Refactoring results in easier to modify tests.

At Pivotal: Developers use a combination of TDD and BDD. While each project is different, programmers tend to use BDD to describe interactions between the user and the system and TDD at a unit test level. Teams use a variety of TDD strategies including testing the responsibilities and interactions [10] or contract testing using mocks [20]. In Pivotal's ideal, the design emerges from the creation and exploration of the test cases.

Anti-pattern: Without this testing practice, developers no longer have the confidence to change any part of the code as they may unknowingly end-up breaking something else.

Alternatives: For a system without a test suite documenting the system specification, a possible remedy is for developers to own particular parts of the system in order to understand the ramifications of changes. Creating strong code ownership and knowledge silos is exactly the problem that sustainable software development is trying to solve.

Writing tests after the code is written could produce a safety net for refactoring, provided that tests correctly exercise the system. (A test that never failed might not be testing anything). We did not observe this behavior and future research is necessary to determine if any testing approach is sufficient for sustainable software development.

4.4.2 Continuous Refactoring

Description: Continuous Refactoring is the systematic improvement of the code base concurrently with new feature development. When developers identify something wrong such as a code smell, they simply fix it. In this regard, developers are caretaking the code by continuously improving it. This practice results in an emergent software design, as well as empathy for the code as developers learn to "listen to the code."

Purpose: Continuous Refactoring enables any pair to work on any part of the system. Long-term benefits for the team include increased code discoverability, code readability, code modifiability, and code simplicity.

At Pivotal: Developers typically do some refactoring while implementing stories. Developers are encouraged to improve the code's design, make the code easier to understand, and increase the discoverability of a component based on its responsibility. Usually, the team prefers "pre-factoring" where the developer does the complicated work to make the implementation of the current story as simple and easy as possible, as opposed to "post-factoring" where refactoring happens after the story is done, but before it is delivered.

Anti-pattern: Removing this practice might produce difficult to modify and messy code. Developers might not be able to easily work on any part of the code base. When refactoring is skipped, code might be simply bolted on to the existing design. Soon it becomes increasingly difficult to bolt more code on. A dilemma arises for the programmers working on the next story: do they continue bolting on more code, or do they perform the pretermitted refactorings? Removing this practice may also result in hard-to-change tests.

Alternatives: Postponing refactoring may be necessary in extreme situations, for instance, when the company might go out of business unless the company releases the next version. In such situations, the team risks taking on uncontrolled technical debt as "refactoring later" turns into "refactoring never."

4.4.3 Live on Master

Description: Live on master means that developers integrate their code several times a day, as quickly as possible. ExtremeProgramming.org calls this practice "Integrate Often" [29].

Purpose: For teams to continuously refactor and minimize the waste of merge conflicts, the entire team needs to routinely merge their code onto master. If a pair communicates to the team that they are actively "refactoring" a component, they are asserting exclusive temporary ownership over the file to avoid merge conflicts. While this is a normal practice for a few hours, if it happens for multiple days, the team is losing collective ownership of that code. The team is not able to receive any of the benefits until the work is merged back to master.

At Pivotal: In the ideal workflow, developers merge their code to master many times a day. If a pair has not merged to master by the afternoon, the pair typically starts examining why this is difficult and explores ways of incrementally making changes. Developers may use branches to save spikes. When rotating pairs, developers may use branches to move work-in-progress code between machines.

Anti-pattern: Removing this practice means that code lives in branches for days or weeks. Integrations might be painful due to merge conflicts and developers might delay needed refactorings. If a developer has code only on their machine, then no one else on the team can use or modify that code. When code lives only on one machine for many days in a row, the machine acts as a "virtual branch." Run-

ning a Continuous Integration box and having long running branches is an anti-pattern.

5. THEORY EVALUATION

Charmaz identifies four criteria for evaluating a Grounded Theory: credibility ("Is there sufficient data to merit claims?"), originality ("Do the categories offer new insights?"), resonance ("Does the theory make sense to participants?"), and usefulness ("Does the theory offer useful interpretations?") [26].

Credibility: The current data set is rich and its analysis leads to theory saturation. (Saturation means that the properties of the theory are complete and are not affected by new data.) The data set comprises 21 intensive interviews conducted in four different offices, field notes from participant observation on Project Quattuor, and the first author's involvement in four other projects as participant-observer.

Originality: The theory uniquely depicts the principles, policies, and practices enabling software development sustainability in an organization. Since the organization under study follows Extreme Programming, it is not surprising that many of the practices of Sustainable Software Development are defined in Extreme Programming. However, overlapping pair rotation and its supporting principles, policies, and practices are central and unique to the proposed theory.

Resonance: The participants examined the theory. The theory resonates with their experience and reflects the way they work.

Usefulness: The theory informs Pivotal engineers as to why Pivotal purposefully avoids knowledge silos, and how the theory's principles, policies, and practices work together to accomplish the team's goals. The theory explains why the principles, policies, and practices should be incorporated together. A few managers use the theory to help potential clients understand how Pivotal achieves the business goals of both the client and Pivotal.

6. THREATS TO VALIDITY

6.1 External Validity

Generalizability across situations: Grounded Theory does not support statistical generalization from a sample to a population. The results may not be applicable to other teams or other domains. There are four broad types of scientific generalization: 1) from data to descriptions, 2) from descriptions to concepts, 3) from concepts to theory, 4) from theory to description [16]. Grounded Theory research involves the first three kinds of generalization. Generalizing from a theory tested in one context to descriptions of a new context (the fourth kind of generalization) could be done by the researchers in the new context, on a case-by-case basis. However, we have not attempted to perform any type four generalizations at this time.

6.2 Internal Validity

Researcher bias: A risk of the participant-observer technique is that the researcher may lose perspective and become biased by being a member of the team. An outside observer might see something the researcher missed. We mitigated this risk by recording interviews and with a colleague reviewing the coding process.

Prior knowledge bias: With Grounded Theory, prior knowledge can aid the researcher in looking at interesting research questions or create difficulties by blinding the researcher about possible explanations [13]. We mitigated this risk with a colleague reviewing the coding process.

7. FUTURE RESEARCH

We are interested in the tension between individual and team ownership, as well as the factors that foster and decrease the sense of ownership. Developers, interaction designers, and product managers all have different goals for their role. In future work, we plan to examine how the sense of ownership is driven by different factors for each role.

Some programmers naturally adapt to team code ownership, while others struggle with the transition. Future research could follow new Pivotal engineers and examine their journey in transitioning from individual code ownership to team code ownership. Perhaps there are specific practices that Pivotal or the development team could employ to ease the transition. We could also investigate the optimal team size for team code ownership, or explore whether Sustained Software Development works for a distributed team with a Shared Schedule.

8. CONCLUSIONS

This paper introduces a descriptive theory of "Sustainable Software Development" as a solution to the challenge of software development sustainability for an ever changing workforce. The theory emerged from a Constructivist Grounded Theory research study. By collecting data from 21 intensive interviews conducted in four different Pivotal offices, field notes from participant-observation on the Project Quattuor, and the first author's involvement in four other Pivotal projects as participant-observer, the study investigates the research question "How do the observed teams develop software effectively while overcoming team disruption?"

The emergent theory is characterized by a collection of synergistic principles, policies, and practices encouraging a positive attitude towards team disruption, knowledge sharing and continuity, as well as caring about code quality. The theory refines and extends Extreme Programming by adding principles, policies, and practices (including Overlapping Pair Rotation) and aligning them with the business goal of sustainability.

Conventional wisdom says that team disruptions should be avoided, and that extensive documentation is needed to prevent knowledge loss during team churn. Unfortunately, documentation often quickly becomes out-of-date and unreliable. The theory positions team code ownership with overlapping pair rotation and knowledge pollination as an alternative and potentially more effective strategy to mitigate against knowledge loss.

The primary benefits to the software developer are the ability to understand the entire system, the ability to work on every story, increased in teaching opportunities to share one's expertise, and more nuanced understanding of the utilized technologies.

The primary benefit to the employer is business agility. The engineering team continues to deliver software week after week, month after month, while surviving cataclysmic events. Things do not fall apart when the superstar developer leaves because features or components are not critically tied to a particular individual. Critical feature work can

be parallelized since anyone can work on any feature. The whole team's talents are leveraged.

9. ACKNOWLEDGEMENT

Thank you to Rob Mee, David Goudreau, Ryan Richard, and Zach Larson for making this research possible. Thank you to Karina Sils for creating Figure 1 and Figure 2 using Sketch.

10. REFERENCES

[1] N. Alaverdyan. Pair programming matrix / board, 2010. URL: http://alaverdyan.com/readme/2010/12/pair-programming-matrix-board/.

[2] K. Awati. Increasing your team's bus factor, 2008. URL: https://eight2late.wordpress.com/2008/09/03/increasing-your-teams-bus-factor/.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[4] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2011.

[6] K. Charmaz. *Constructing Grounded Theory*. SAGE Publications, 2014.

[7] J. Chong. Social behaviors on XP and non-XP teams: A comparative study. In *Proceedings of the Agile Development Conference*, ADC, Washington, DC, USA, 2005. IEEE Computer Society.

[8] J. O. Coplien. A generative development process pattern language. In *Proceedings of Pattern languages of Program Design*, PLoP, 1994.

[9] R. Davies and L. Sedley. *Agile Coaching*. Pragmatic Bookshelf, 2009.

[10] S. Freeman and N. Pryce. *Growing object-oriented software, guided by tests*. Pearson Education, 2009.

[11] B. Glaser. *Theoretical Sensitivity: Advances in the Methodology of Grounded Theory*. Sociology Press, 1978.

[12] B. Glaser. *Basics of grounded theory analysis: emergence vs forcing*. Sociology Press, 1992.

[13] B. Glaser. *Doing Grounded Theory: Issues and Discussions*. Sociology Press, 1998.

[14] D. Izquierdo-Cortazar, G. Robles, F. Ortega, and J. M. Gonzalez-Barahona. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *42nd Hawaii International Conference on System Sciences*, HICSS, 2009.

[15] D. Joseph, K.-Y. Ng, C. Koh, and S. Ang. Turnover of information technology professionals: A narrative review, meta-analytic structural equation modeling, and model development. *MIS Quarterly*, Sept. 2007.

[16] A. S. Lee and R. L. Baskerville. Generalizing generalizability in information systems research. *Information Systems Research*, 2003.

[17] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 2003.

[18] S. McConnell. Managing technical debt. Technical report, Construx Software Builders, Inc, 2008.

[19] G. Paci. Trucknumber. Portland Pattern Repository. URL: http://c2.com/cgi/wiki?TruckNumberFixed.

[20] J. Rainsberger. Integration tests are a scam, 2013. URL: https://www.youtube.com/watch?v=VDfX44fZoMc.

[21] P. Ralph. Developing and evaluating software engineering process theories. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE, 2015.

[22] F. Ricca, A. Marchetto, and M. Torchiano. On the difficulty of computing the truck factor. In *Product-Focused Software Process Improvement*. Springer, 2011.

[23] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus. Quantifying and mitigating turnover-induced knowledge loss: Case studies of chrome and a project at Avaya. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE, 2016.

[24] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, 2001.

[25] T. Sedano, P. Ralph, and C. Péraire. Practice and perception of team code ownership. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE, 2016.

[26] K.-J. Stol, P. Ralph, and B. Fitzgerald. Grounded theory in software engineering research: A critical review and guideline. In *Proceedings of the 2016 International Conference on Software Engineering*, ICSE, 2016.

[27] B. W. Tuckman. Developmental sequence in small groups. *Psychological bulletin*, 1965.

[28] J. Vanhanen and H. Korpi. Experiences of using pair programming in an agile project. In *40th Annual Hawaii International Conference on System Sciences*, HICSS, 2007.

[29] D. Wells. Integrate often, 1999. URL: http://www.extremeprogramming.org/rules/integrateoften.html.

[30] L. Williams and R. Kessler. *Pair Programming Illuminated*. Addison-Wesley Pearson Education, 2002.

[31] F. Zieris and L. Prechelt. Observations on knowledge transfer of professional software developers during pair programming. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016.