Sustainable Software Development: Evolving Extreme Programming

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Todd Sedano

B.S., Mathematics and Computer Science, Carnegie Mellon University
M.S., Software Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May 2017

Acknowledgements

I want to thank my committee members, Cécile Péraire, Paul Ralph, Martin Griss, and Jim Morris, for their guidance and expertise.

Cécile, you have been a wonderful advisor. Thank you for all the meetings and discussions. You have shaped me as a researcher. I continue to be impressed by your ability to structure a narrative. I am grateful that you allowed me to follow wherever the research was leading. Another advisor might have forced me to stay with a particular research question. Instead, our research and results are much more powerful and relevant to industry.

Paul, I am thankful to have you as co-advisor, guiding me through the Byzantine ways of academia. On some days, writing code is much simpler than navigating the waters of publishing research. I appreciate your deep understanding of theory, research methods, and scholarship. I continue to be envious of your diction, but I have learned to be even more grateful for our friendship.

Martin, I appreciate your mentoring and leadership. I coach others in the kind way that you coached me. Above all, I appreciate your strong commitment to ethical living. There were times where others would have compromised, and you surprised me by staying true to your principles.

Jim, I am grateful that you started the Silicon Valley campus of Carnegie Mellon University. I remember so many pleasant memories of working at Carnegie Mellon. When my first advisor did not want to work with a part-time student, you were there to help me through a difficult transition.

I wish to thank my family for their support of my graduate courses, research, and traveling to conferences. Without my wife Karie, this would have been impossible. Thank you Claire and Colette for welcoming me home each day by running out the front door.

I wish to thank the leadership of Pivotal for making this research possible. Thank you to Rob Mee, David Goudreau, Ryan Richard, Zach Larson, Elisabeth Hendrickson, and Michael Schubert for making this research possible.

Thank you to Karina Sils for creating Figure 6.1 and Figure 6.2 using Sketch. Thanks to Ben Christel for his assistance in sorting retro topics and helping with the initial analysis for Software Engineering Waste.

The source of support for this research is self-support with tuition reimbursement from Carnegie Mellon University and Pivotal.

Abstract

Context: Software development is a complex socio-technical endeavor that involves coordinating different disciplines and skill sets. Practitioners experiment with and adopt processes and practices with a goal of making their work more effective.

Objective: To observe, describe, and analyze software development processes and practices in an industrial setting. Our goal is to generate a descriptive theory of software engineering development, which is rooted in empirical data.

Method: Following Constructivist Grounded Theory, we conducted a 2.5 year participant-observation of eight software projects at Pivotal, a software development company. We interviewed 33 software engineers, interaction designers, and product managers, and analyzed one year of retrospection topics. We iterated between data collection, data analysis and theoretical sampling until achieving theoretical saturation and generating a descriptive theory.

Results: 1) This research introduces a descriptive theory of Sustainable Software Development. The theory encompasses principles, policies, and practices aiming at removing knowledge silos and improving code quality, hence leading to development sustainability. 2) At the heart of Sustainable Software Development is team code ownership. This research widens our understanding of team code ownership. Developers achieve higher team code ownership when they understand the system context, have contributed to the code in question, perceive code quality as high, believe the product will satisfy the user needs, and perceive high team cohesion. 3) This research introduces the first evidence-based waste taxonomy, identifying eight wastes along with causes and tensions, and compares it with Lean Software Development's waste taxonomy.

Conclusion: The Sustainable Software Development theory refines and extends our understanding of Extreme Programming by adding principles, policies, and practices (including Overlapping Pair Rotation) and aligning them with the business goal of sustainability. One key aspect of the theory is team code ownership, which is rooted in numerous cognitive, emotional, contextual and technical factors and cannot be achieved simply by policy. Another key dimension is waste identification and elimination, which has led to a new taxonomy of waste. Overall, this research contributes to the field of software engineering by providing new insights, rooted in empirical data, into how a software organization leverages and extends Extreme Programming to achieve software sustainability.

Contents

List of Tables

List of Figures

Abstract

Context: Software development is a complex socio-technical endeavor that involves coordinating different disciplines and skill sets. Practitioners experiment with and adopt processes and practices with a goal of making their work more effective.

Objective: The purpose of this research is to observe, describe, and analyze software development processes and practices in an industrial setting. Our goal is to generate a descriptive theory of software engineering development, which is rooted in empirical data.

Method: Following Constructivist Grounded Theory, we conducted a two-year five-month participant-observation of eight software development projects at Pivotal, a software development company whose process is based on Extreme Programming. We interviewed 33 software engineers, interaction designers, and product managers, and analyzed one year of retrospection topics. We iterated between data collection, data analysis and theoretical sampling until achieving theoretical saturation and generating a theory for understanding.

Results: 1) This research 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. 2) At the heart of Sustainable Software Development is team code ownership. This research widens the current definition and understanding of team code ownership. It identifies five factors that affect ownership. Developers achieve higher team code ownership when they understand the system context, have contributed to the code in question, perceive code quality as high, believe the product will satisfy the user needs, and perceive high team cohesion. 3) This research introduces the first evidence-based waste taxonomy, identifying nine wastes along with causes and tensions within wastes. It also provides a comparison with the taxonomy of wastes found in Lean Software Development.

Limitations: While the results are highly relevant to the observed company, Pivotal, the outcomes might not apply to organizations with different software development cultures.

Conclusion: The Sustainable Software Development 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. One key aspect of the theory is team code ownership, which is rooted in numerous cognitive, emotional, contextual and technical factors and cannot be achieved simply by policy. Another key dimension is waste identification and elimination, which has led to a new taxonomy of waste. Comparing this taxonomy to Lean Software Development's list of wastes revealed our taxonomy's parsimony and expressiveness while illustrating wastes not covered

by previous work. Overall, this research contributes to the field of software engineering by providing new insights, rooted in empirical data, into how a software organization leverages and extends Extreme Programming to achieve software sustainability.

Chapter 1

Introduction

Imagine that you are a Vice President of a multi-million dollar software development business unit managing 100 software engineers and 20 product managers. You have 25 teams working on various interrelated software products. The entire organization balances 1) adding new features to the next release, 2) solving customer escalations from installed products, and 3) handling maintenance chores such as routinely upgrading technology dependencies. Over the last few releases, you have seen a continuing decrease in your teams' productivity, and a steep increase in complaints from customers related to product quality. And somehow, there is never enough time to address those challenges.

In the middle of all this, one team is extremely difficult to manage. The team is composed of highly skilled software engineers. They write well tested, clean code. The test suite is slow, but is thorough and catches issues before they get to the customer.

The team resists collaborating with other teams or customers. The team developed an "us versus them" or "us versus management" attitude. Their product manager would like the team to be in the office more often to increase collaboration. While the organization expects teams to be in the office during core business hours, this team believes that as long as the work gets done, nobody should care where or when they accomplish the work. The product manager is growing frustrated with the team's attitude about features. Their focus appears to be on delivering something that works without truly understanding how the features should work. As a result, when they deliver code, the implementation does not always solve the customer's needs or works the way the customer expects.

How did software development come to this? Why is software development so hard? Why is collaboration so difficult? How can we build teams that embrace collaboration, help others solve the problems, and train others to do their work?

The complexities of software development emerge, in part, due to 1) uncertainty about product to market fit resulting in changes to the feature set during and after development, 2) constantly changing technologies, 3) the challenge of maintaining legacy code, 4) the lack of physical constraints resulting in boundless solutions, 5) measuring progress is challenging because project scope is a moving target and bringing visibility to an intangible process is difficult, and 6) software is written by people having various expertises as well as social and technical skills. Teams require coordination and collaboration. Building high performing teams requires effort and care. Unlike assembly line work, engineers are not fungible. Even if two engineers have the same competency in a set of technologies, their individual temperaments and personalities affect team dynamics. Building software is challenging.

While it appears that any set of processes will work for tiny teams in the short term, creating an engineering culture where large teams deliver value week after week, month after month regardless of team disruption and changing circumstances is no easy task. Any company that achieves proficiency in software development becomes an interesting research opportunity. We choose Pivotal because it is successful; it is unusual in its continued use and evolution of Extreme Programming [8]; it is accessible and cooperative with research. Chapter 3 describes Pivotal, our research context.

In order to understand how Pivotal develops software, we employed Constructivist Grounded Theory as the research method. Chapter 4 describes Constructivist Grounded Theory and Chapter 5 describes our data sources and particular use of Grounded Theory. This study had an unusually ambitious scope. In many Grounded Theory studies, the researcher spends a few weeks interviewing participants and analyzing data. Here, one researcher spent two years and five months working with participants 5 days per week on eight teams.

Applying Grounded Theory at Pivotal discovered the theory of Sustainable Software Development, an explanation on how Pivotal teams can thrive despite team disruption and successfully deliver software to their stakeholders. Chapter 6 describes the theory's principles, policies, and practices. The theory explains how teams survive disruption by actively removing knowledge silos and caretaking the code. At the heart of the Pivotal process is team code ownership. Many of Pivotal's practices have the effect of promoting team code ownership while removing individual code ownership. Chapter 7 identifies five factors that affect the team's sense of code ownership and explains why psychological ownership causes some engineers to struggle with the transition from individual code ownership to team code ownership.

The Grounded Theory study also identified that Pivotal teams actively detect and remove waste. Since software development is an organic and messy problem space, finding all kinds of waste is no surprise. Chapter 8 conducts the first empirical research study into a waste taxonomy and contrasts it with Lean Software Development's waste taxonomy.

Chapter 9 discusses future research and concludes the research study.

Appendix A provides the chain of evidence for the waste taxonomy after iteratively applying constant comparison.

Appendix B showcases interview data. The initial interviews began with the question, "draw your view of Pivotal's software development process." The appendix includes a sample of these drawings with interview transcript snippets where the interviewee describes the drawing. Once team code ownership emerged as a core category, many interviews began with the question, "draw how you feel about the code." Section B.7 provides some of these drawings. When the pictures are not obvious, a brief narrative explains the illustration.

Chapter 2

Extreme Programming

Kent Beck invented Extreme Programming while running the Chrysler payroll software project in 1996 [7]. In 1999, Beck published Extreme Programming Explained: Embrace Change [7] and updated the book in 2004 [8]. This chapter covers the latest version.

Extreme programming balances business needs of repeatedly and reliably delivering software with the human needs of the participants. Extreme Programming is characterized by a set of values, principles, and practices. When transitioning a team to Extreme Programming, Beck recommends first adopting the primary practices.

2.1 Values

Values help guide a team. Values remind a team why they do certain practices. Without values, a team might "cargo-cult" a practice (e.g. continue to religiously follow a practice without understanding why they follow it.) Beck says "values are the roots of the things we like and don't like in a situation" [8]. Values are abstract concepts, and hard to observe, whereas it is much easier to examine if a team is following a practice. "Practices are evidence of value . . . Just as values bring purpose to practices, practices bring accountability to values" [8].

Communication: Extreme programming is a collaborative endeavor with frequent communication. Teams prefer open and transparent communication. "You can listen to people who have had similar problems in the past" [8]. When combined with the practice of reflection, the team can ask "what communication do you need to keep yourself out of this trouble in the future?" [8]

Simplicity: Teams prefer simpler solutions to more complex solutions. Teams strive to "make a system simple enough to gracefully solve only today's problem" [8]. Simpler solutions often take more work to discover and implement than complex solutions, but simpler solutions often require less communication than complex solutions.

Feedback: Feedback provides an opportunity for the team to respond to changing circumstances in the marketplace, in the product, in the team, and with the development process. Change requires feedback. In order for continuous improvement to work, the team decides to collect feedback at multiple frequencies and at multiple levels. The team prefers incremental improvement over expecting perfection. Teams attempt to shorten the feedback cycle so that the team can respond sooner rather than later.

Courage: "Courage is effective action in the face of fear" [8]. Sometimes it takes courage to make the needed change. Messy code will remain messy unless a programmer demonstrates courage to improve it. Individuals consider the consequences before making a change. With change, the team accepts the risk of unintended consequences (e.g. the tests fail.) When the team knows the problem, courage is a bias to action. When the team does not know the underlying problem, courage may simply be patience.

Respect: Collaborative software development is founded on respect. Team members decide to respect each other. "If members of a team don't care about each other and what they are doing, XP won't work" [8].

Others: Teams may adopt additional values and align their practices around the values.

2.2 Principles

The principles help guide a team in implementing the values. When two potential practices achieve the same value, the principles guide the team. Beck shows that both documentation and daily conversations satisfy the need of communication, yet the principle of humanity reminds that daily communication satisfies the psychological need of human connection [8].

Humanity: The practices that a team follows need to satisfy human needs. Alternative software development practices may not respect psychological needs. A practice may grind away at an individual's humanity [8]. Beck lists fundamental human needs as basic safety, accomplishment, belonging, growth, and intimacy [8]. Team software development balances the needs of the individual with the needs of the team.

Economics: The practices that a team follows need to balance business needs with technical needs. People buying the product fuels the business. "Software development is more valuable when it earns money sooner and spends money later" [8]. Incremental design and incremental delivery allow the team to release earlier and delay expenses until later. Teams avoid building the illusive "perfect" product.

Mutual Benefit: The practices that a team follows need to find solutions that benefit everyone involved. A practice that does not benefit the team now is not mutually beneficial, e.g. writing documentation for a faceless, future maintenance team. Mutual beneficial practices help people now and in the future.

Self-Similarity: The practices that a team follows can potentially be adapted to different contexts. Try applying solutions that work in one context to another context. Beck sees a similarity in listing themes for a quarter, addressing stories in a week, writing tests for a story.

Improvement: The practices that a team follows evolve and improve over time. Teams benefit from constant improvement. The point of XP is "excellence in software development through improvement" [8]. Starting is preferred to waiting for perfection.

Diversity: The practices that a team follows should resolve conflict productively. "Two ideas about a design present an opportunity, not a problem" [8]. Multiple perspectives provides richer solutions. Teams benefit from a diversity of skills, perspectives, attitudes, and experiences.

Reflection: The practices that a team follows should enable reflection at different frequencies and multiple levels. Teams reflect on how and why they are working. The team sees mistakes as opportunities for growth.

Flow: The practices that a team follows should decrease batch sizes to enhance flow. "Flow in software development is delivering a steady flow of valuable software by engaging in all the activities of development simultaneously" [8]. Teams want a steady stream of work moving through the system. The ideal "batch size" is one story per developer. Anything that interrupts the flow of work needs removing.

Opportunity: The practices that a team follows need to frame problems as opportunities. A team "playing it safe" will make fewer mistakes but also move slower than needed.

Redundancy: The practices that a team follows should address difficult problems in multiple ways. "The critical, difficult problems in software development should be solved several different ways" [8]. For example, XP handles the critical, difficult problem of defects with "pair programming, continuous integration, sitting together, real customer involvement, and daily deployment" [8].

Failure: The practices that a team follows should bias a team towards action. When the team does not know what to do, try something and get feedback. "If you're having trouble succeeding, fail" [8]. Sometimes replacing lengthy conversation and debate with trying multiple ideas produces less waste.

Quality: The practices a team follows should deliver high-quality products. Increases in quality "leads to improvements in other desirable project properties, like productivity and effectiveness" [8]. Beck argues that short-changing quality is not a way to make a project go faster.

Baby Steps: The practices a team follows need to prefer small incremental changes. Start with the simplest smallest change possible and iterate from there. Decompose significant changes into small in-

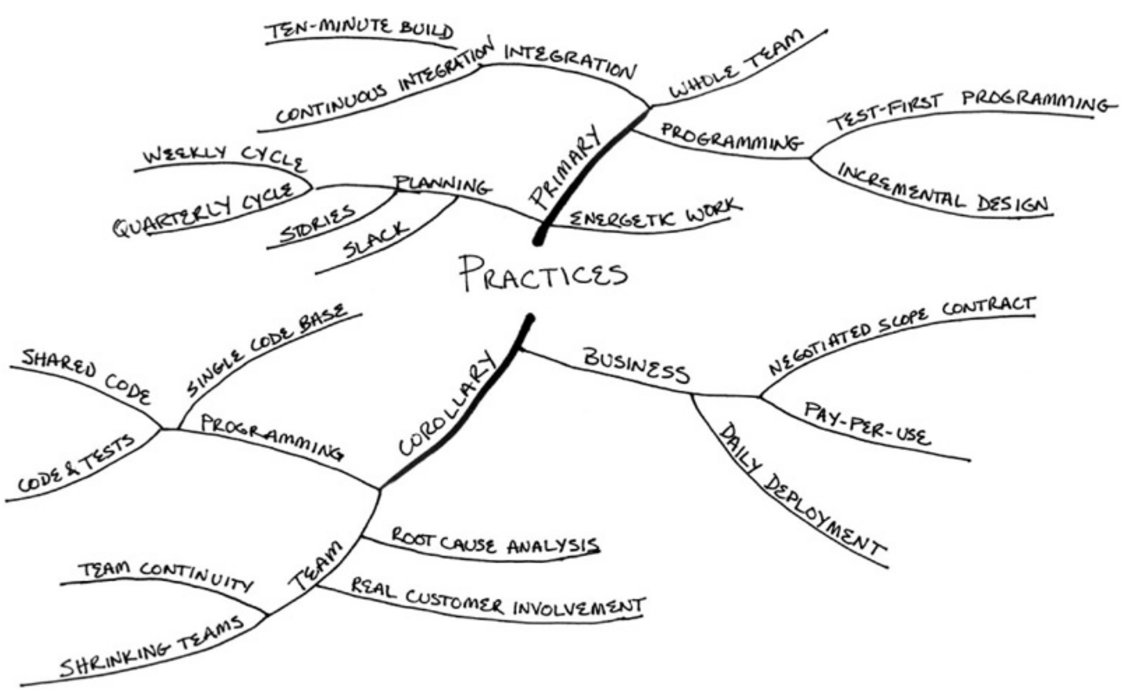


Figure 2.1: Extreme Programming Practices [8]

cremental changes. Beck often asks, "what's the least you could do that is recognizably in the right direction?" [8].

Accepted Responsibility: The practices a team follows should empower the people working on a problem with the authority to solve it. Responsibility is accepted, not assigned. Avoid the situation where people in one part of the organization are telling others what to do.

2.3 Primary Practices

For a team transitioning to Extreme Programming, Beck recommends starting with the primary practices. Beck claims that "the primary practices are useful independent of what else you are doing" resulting in an immediate improvement, whereas the corollary practices work well once a team has implemented the primary practices. Extreme programming is not dogmatic about the list of practices. The team chooses which practices to apply.

Sit Together: Software development is a collaborative endeavor. Co-locating teams increases collaboration. Distributed teams are still possible, yet "Sit Together predicts that the more face time you have, the more humane and productive the project" [8].

Whole Team: Whole team is also known as cross-functional teams. Include people on the team with the skills needed for success. Beck points out that a sense of team fulfills an individual's psychological needs. "We belong. We are in this together. We support each others' work, growth, and learning." [8]

Informative Workspace: "An interested observer should be able to walk into the team space and get a general idea of how the project is going in fifteen seconds" [8]. The team can modify the workspace to make it more conducive for their work.

Energized Work: "Work only as many hours as you can be productive and only as many hours as you can sustain" [8]. Working longer hours may make the team less productive. Team members need to take care of themselves; this includes staying home to rest when sick. The work patterns of the team should be sustainable in the long run.

Pair Programming: Every part of a production system is written by two programmers sitting side by side. The pair works together to solve problems, keep each other on task, learn from each other, and abide by the team's practices.

Stories: Stories are "units of customer-visible functionality" [8]. Stories have a name and a description or graphical depiction. Stories track the work to be done.

Weekly Cycle: Each week, the team accepts a certain amount of work to accomplish. During a meeting at the beginning of the week, the team reviews progress to date, the customer chooses a week's worth of stories, and the engineers decompose the stories into tasks and individually estimate each task. "The goal is to have deployable software at the end of the week which everyone can celebrate as progress" [8].

Quarterly Cycle: Each quarter, the team identifies the stories for the next quarter. During a meeting, the team identifies bottlenecks, initiates repairs, plans the feature sets for the quarter, and selects a quarter's worth of stories.

Slack: During the weekly cycle meeting, include tasks in the plan that the team can drop if the team gets behind. Adding slack to the schedule enables a sustainable work environment.

Ten-minute Build: Building the code and running the test suite should be automated and short. Longer builds increase the feedback loop.

Continuous Integration: When working on new features, routinely integrate new code into the master code repository. Avoid long-lived branches. Beck suggests un-integrated code live on its own "no more than a couple of hours" [8]. The longer the team waits to integrate, the more unpredictable and expensive integration becomes.

Test-First Programming: The developers write tests before writing code. Every production line of code is preceded by a test. Test-first programming addresses several concerns: scope-creep, coupling and cohesion, trust, and rhythm. The practice mitigates against scope-creep; without a clear goal, a programmer can easily write more code than is necessary. The practice improves coupling and cohesion as easy-to-test code often has beneficial design qualities. The practice builds trust in the code as engineers trust tested code more than un-tested code. The practice produces rhythm as it provides a cadence of accomplishing goals.

Incremental Design: Build the system incrementally and evolve the design every day. In the past, software teams might often had long design phases that assumed features would not change. When the team's understanding of the system changes, make incremental changes to the system to align the code's design with the team's desired goal. Teams routinely remove duplication.

2.4 Corollary Practices

Once a team has implemented the primary practices, then the team can start implementing the corollary practices. Beck says that it seems "difficult or dangerous to implement [the corollary practices] before completing the preliminary work of the primary practices" [8].

Real Customer Involvement: Make the customer a part of the team. "The point of customer involvement is to reduce wasted effort by putting the people with the needs in direct contact with the people who can fill those needs" [8]. Even if a customer is busy, involving them whenever possible is preferred to not involving them at all. In this situation, the team needs to find alternative ways of validating features with users.

Incremental Deployment: Switch between a legacy system and a replacement system in incremental steps. Avoid a hard switch over from a legacy system to a new system.

Team Continuity: "Keep effective teams together" [8]. People are not fungible, and building relationships is part of creating a high performing team. Rotating people through teams helps spread knowledge.

Shrinking Teams: Try to improve team efficiency so that the team can shrink in size. When a team becomes more productive, re-allocated extraneous team members to other teams.

Root-cause analysis: When a defect occurs, determine why it happened. Modify testing strategy to prevent that kind of mistake from happening again. Write an automated system test and an automated unit test then fix the code.

Shared Code: Everyone on the team can modify any of the system. When engineers sees something broken, they fix it.

Code and Tests: The tests and code are the only permanent artifacts. Everything else (e.g. documentation) is generated from the code. Anything that contributes to what the system does today and what it will do tomorrow is valuable, "everything else is waste" [8].

Single Code Base: Ideally there is only one branch of the code. Multiple branches, (e.g. for releases, or customers) creates extra work. Branches live a few hours at most.

Daily Deployment: Every day put a new version of the code into production.

Negotiated Scope Contract: "Write contracts for software development that fix time, costs, and quality but call for an ongoing negotiation of the precise scope of the system."

Pay-per-use: Pay-per-use provides alignment between revenue and the features that generate the revenue. "Connecting money flow directly to software development provides accurate, timely information with which to drive improvement" [8].

2.5 Criticism of Extreme Programming

An issue with any of the practices typically can be an issue with Extreme Programming. For example, some engineers do not enjoy pair programming as they prefer to work individually. Some engineers prefer to design software that deals with every possible future request. Some engineers prefer not to test their code.

If a team is unable to implement a practice, then it requires modification of Extreme Programming. Difficulty in locating a customer willing to work with the team means that the team will need to adapt Extreme Programming to fit the team's challenges. Organizations with distributed teams will need to adapt the practices. Some software development organizations prefer taking on technical debt so that they can ship as early as possible.

Achieving buy-in with key stakeholders can be challenging. Managers may be skeptical of assigning two development resources to every software story.

2.6 Conclusion

Every year, VersionOne surveys the industry looking for trends in Agile software development [71]. While surveys do have limitations, their data shows that many of the Extreme Programming practices are widely adopted by teams as shown in Figure 2.2. (Note that the list of surveyed practices is different than the practices in Extreme Programming.) One of the harder to adopt practices, pair programming is at 24%. Pivotal provides a rare opportunity to study Extreme Programming in industry.

Extreme programming provides a collection of principles, values, and practices for adopting a balanced approach to software development.

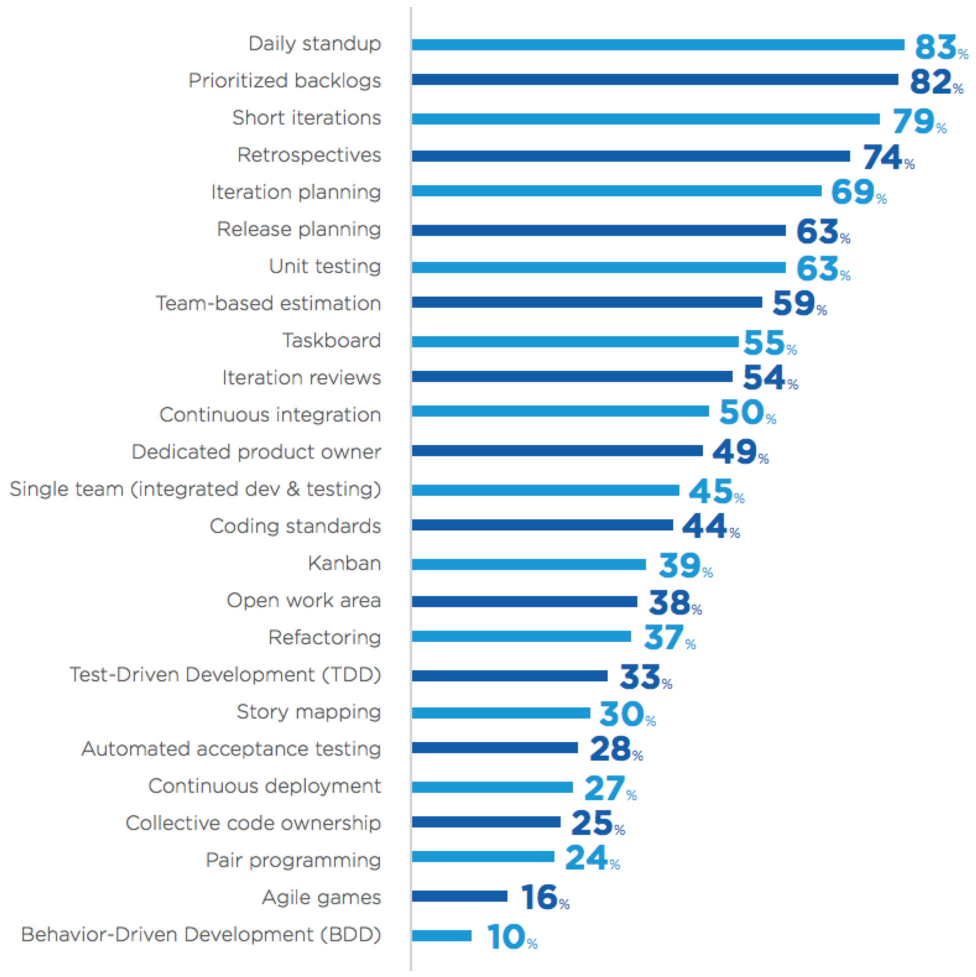


Figure 2.2: Agile Practices Adoption [71]

Chapter 3

Research Context

3.1 Pivotal Labs

Pivotal Labs is a division of Pivotal—a large American software company (with 17 offices around the world). Pivotal Labs provides teams of agile developers, product managers, and interaction designers to other firms. Its mission is not only to deliver highly-crafted software products but also to help transform 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 [8] in an environment conducive to agile development.

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.

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.

Pivotal Labs has followed Extreme Programming [8] 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.

3.2 Day in the Life

This section presents a typical day in the life of a Pivotal software engineer.

Pivotal provides breakfast to promote engineers starting the day at the same time. Many engineers socialize with their coworkers while eating breakfast.

Office stand-up begins at 9:06 am [61]. Everyone gathers around in a large circle. The stand-up covers "new faces," "helps," "interestings," and "events." The office stand-up lasts a few minutes. The two people running stand-up send an email to the company with a summary of the brief meeting. People then disperse for their team's stand-ups.

Team stand-ups provide a synchronization point for the team. Small teams (under eight people) typically will review what each individual accomplished yesterday and discuss any blockers to finishing the work. Large teams discuss "interestings" and "helps." Some teams use a whiteboard to track "parking lot" issues (items needing discussion during stand-up). After team stand-up, the engineers then decide "pairing." For work in progress, each pair decides who will continue with it. The available engineers then re-pair with engineers continuing the work. If a pair has nothing to work on, they take the next story at the top of the backlog. The team then determines which pairs will use which computer. If there is work already in progress on one machine, the pair continues the work on the same machine. The team configures the development environments to be identical on all machines. Pairing continues until 12:30 pm when the team takes a lunch break. After lunch, pairing continues until 6:00 pm.

While coding, the developers follow Test Driven Development (or Behavior Driven Development.) Tests are written first before determining the design or writing code. A failing test provides the developers with a short term goal. The rhythm is to start a story, refactor if necessary, write a small failing test, write just enough code to get the test to pass, refactor if necessary, and repeat by writing a small failing test. The ideal flow is quick, short cycles of Test Driven Development. When a pair finishes their work, they start the next story at the top of the backlog.

The interaction designer validates product features with user research, creates mock-ups of the user interaction, and validates mock-ups and the implemented product with users. After validating the mock-ups, the product manager converts the new feature into a set of small stories. Each story represents a small piece of interaction between the user and the system. By decomposing a feature into granular stories, the product manager can identify high value work to start now and delay low value work.

The product manager determines the features and the sequence of stories. The prioritization is kept in a backlog. The product manager decides "what" the product should do. The engineers determine the implementation details, "how" the code should be written to accomplish the task in the story. The product manager changes the direction of the product at any point in time by resequencing the backlog. The product manager does not change any "in-flight" stories (stories that the developers are working on).

During each week, the team holds an iteration planning meeting. The product manager and designer communicate to the engineers the upcoming work, building a shared understanding of the stories. The engineers communicate to the product manager the complexity and risk associated with each story.

At the end of each week, the team holds a retrospection meeting to examine what is working well for the team, what needs improvement, and determine any action items to accomplish the identified improvements.

Chapter 4

Grounded Theory

Constructivist Grounded Theory [11] provides an iterative approach to data collection, data coding, and analysis resulting in an emergent theory.

Grounded Theory immerses the researcher within the context of the research subject from the point of view of the research participants. As the research progresses, Grounded Theory allows the researcher to incrementally direct the data collection and theoretical ideas. The process provides a starting place for inquiry, not a specific goal known at the beginning of the research. As the researcher interacts with the data, the data influences how the research progresses and guides the research direction. When starting a Grounded Theory research study, the core question is, "what is happening here?" [19]. The researcher can apply this question to a domain of study, for example, "what is happening at the studied organization when it comes to software?"

Charmaz encourages the researcher to follow this Grounded Theory strategy: "seek data, describe observed events, answer fundamental questions about what is happening, and then develop theoretical categories to understand it" [11].

Grounded Theory follows an iterative process as illustrated in Figure 4.1 [68]. After identifying a population to study, data is collected from interviews, participant observation, ethnographic study, or existing documents. Each kind of data, such as interviews, are coded and analyzed using constant comparison for the purpose of generating insights and recording memos. The coding process labels data with tags that emerge from the data. Constant comparison compares codes to codes for the purpose of identifying relationships between the codes. Memos are researcher diary notes. The principal activity is creating memos, and it supersedes and interrupts all other activities. The researcher sorts memos into a paper draft with data collection and analysis continuing until theoretical saturation occurs. If a new avenue of research arises, additional data collection techniques are sometimes useful for generating the needed

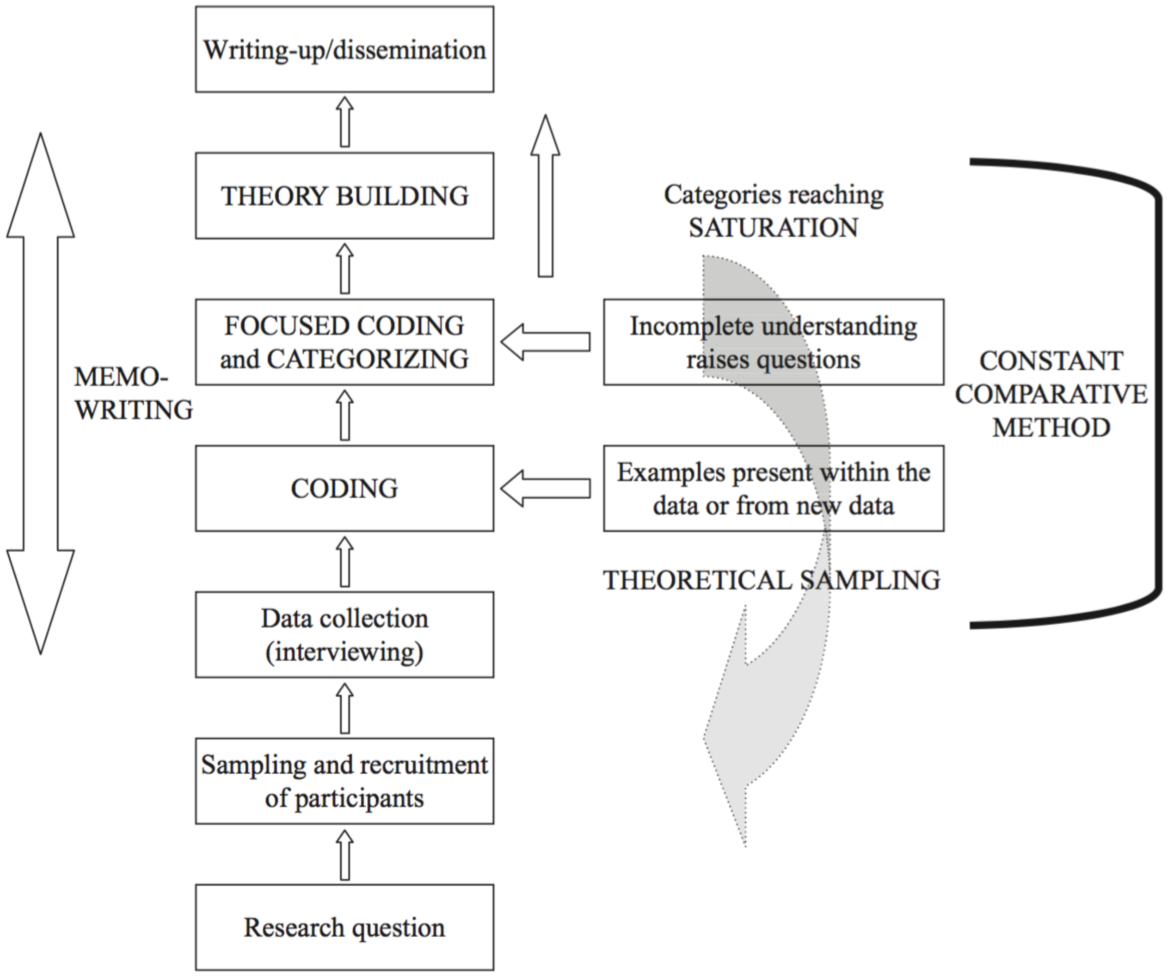

Writing-up/dissemination
THEORY BUILDING
FOCUSED CODING and CATEGORIZING
CODING
Data collection (interviewing)
Sampling and recruitment of participants
Research question
MEMO-WRITING
Categories reaching SATURATION
Incomplete understanding raises questions
Examples present within the data or from new data
THEORETICAL SAMPLING
CONSTANT COMPARATIVE METHOD

Figure 4.1: Visualization of Grounded Theory by Tweed and Charmaz [68]

data. The data advances from the initial codes to focused codes, focused codes to core categories, and core categories to an emergent theory.

Grounded Theory allows the addition of new aspects of the research while gathering data, which can even happen late in the analysis. The research question guides the data collection process, and the accumulation of knowledge alters the data collection process. It is common for early research to illuminate new angles. “Grounded Theory can give you flexible guidelines rather than rigid prescriptions” [11]. The researcher’s guiding interests can be used “as “points of departure” to form interview questions, to look at data, to listen to interviewees, and to think analytically about the data” [11].

There are three widely used versions of Grounded Theory: Classic Grounded Theory as championed by Barney Glaser [25], Straussian Grounded Theory as documented by Anselm Strauss and Juliette Corbin [64], and Constructivist Grounded Theory as described by Kathy Charmaz [11].

Barney Glaser and Anselm Strauss invented Grounded Theory while performing the research for the Awareness of Dying book [24]. In response to associates asking for more details about the research process utilized in producing the novel results for Awareness of Dying, Glaser and Strauss wrote The Discovery of Grounded Theory [25] in 1967. In 1988, Strauss and Juliet Corbin published Basics of Qualitative Research [64]. Glaser strongly felt that the book mischaracterized the approach, even asking Strauss to withdraw it from publication. Apparently, both authors had different points of view for what they invented. In a heated response, Glaser published Basics of Grounded Theory Analysis in 1992 [20] clearly differentiating his perspective from Strauss. In time, several researchers evolved the approach using variations (such as recording an interview) that Glaser clearly describes as not part of Classic Grounded Theory. Furthermore, they switched from a positivist point of view to a constructivist perspective. One of these researchers, Kathy Charmaz, a Ph.D. student of Glaser, published Constructing Grounded Theory in 2006 [11].

Stol et al [63] provide an excellent summary of the differences between the three types of Grounded Theory listed in Table 4.1 shown at the end of this section. There are several significant differences which are summarized here:

- The influence of research questions: emerges from the research (Classic Grounded Theory), may be defined upfront (Straussian Grounded Theory), may be defined upfront and evolves through study (Constructivist Grounded Theory)
- The role of existing literature: delayed in the process (Classic Grounded Theory), used when needed (Straussian Grounded Theory, Constructivist Grounded Theory)
- Theoretical Coding (Classic Grounded Theory, Constructivist Grounded Theory) versus Axial Coding (Straussian Grounded Theory)
- Analytic questions: "what is this data a study of?" (Classic Grounded Theory, Constructivist Grounded Theory), asking questions of the phenomena such as why, how, and when (Straussian Grounded Theory)
- Philosophical differences: objectivism (Classic Grounded Theory), pragmatism (Straussian Grounded Theory), and social constructionism (Constructivist Grounded Theory)
- Evaluation criteria

Classic Grounded Theory emphasizes action and process. It looks at the question, "what is going on here?" and decomposes it into "what are the basic social processes?" and "what are the basic social psychological processes?" [11].

Straussian Grounded Theory relies on Axial Coding as a key differentiator from Glaser's Theoretical Coding. Axial Coding is a systematic process for looking at the relationship between codes for aiding in the emergence of the theory. Axial Coding answers the questions "when, where, why, who, how, and with what consequences" [64]. Glaser [20] felt that axial coding favored the six C's (Causes, Contexts, Contingencies, Consequences, Covariances and Conditions) one of the eighteen theoretical families identified in Theoretical Sensitivity without allowing emergence of the other theoretical families [19]. Charmaz prefers keeping codes "simple, direct, analytic, and emergent" instead of applying axial coding which can distract the researcher from understanding what the data means by focusing on the process [11]. Those in favor of Straussian Grounded Theory see Axial Coding as a more systematic process than Theoretical Coding. Constructivist Grounded Theory deviates from Classic Grounded Theory by embracing constructivism. Section 4.12 explores how constructivism affects the research process. In addition to the differences listed above, the interview process is also different.

Constructivist Grounded Theory encourages the researcher to record and transcribes interviews. Glaser feels that this slows down the research process, overwhelms the researcher with too many codes to analyze, and elicits properline data from the interviewee since they know they are recorded. (Properline data is when interviewees provided information that is socially acceptable for delicate matters. Social norms and corporate narratives reinforce properly aligned data. Glaser is much more interested in understanding what is actually happening. [23].) Charmaz describes Glaser and Strauss's interview approach as [25] "smash-and-grab" by prioritizing the needs of the researcher over the needs of the participants. Instead, she recommends allowing the participant to set the tone and pace of the interview with the researcher matching and mirroring the participant's tone and pace. Constructivist Grounded Theory specializes in breaking down phenomena and reflecting on dialog with "what" and "how" questions.

4.1 Interview Technique

Interviews are the most common technique of gathering data in a Grounded Theory study. Grounded Theory interviews are open-ended explorations of the participant's perspective. Ideally, the interviewer does not force topics but merely follows the path of the interviewee. The interview initially starts with broad, open-ended questions to allow the emergence of unexpected narratives.

Charmaz suggests "intensive interviews," which are "open-ended yet directed, shaped yet emergent, and paced yet unrestricted" [11]. Open-ended questions enter into the participant's personal perspective within the context of the research question. The interviewer attempts to abandon assumptions in order to understand and explore the interviewee's perspective. Charmaz [11] contrasts intensive interviews with informational interviews (collecting facts), and investigative interviews (exposing hidden intentions, practices or policies).

The goal is for the researcher to understand the participant's views and actions. The researcher does not need to agree with those views and actions, just interpret them. The researcher can discover what the participants take for granted.

During the interview, the constructivist explores areas of theoretical interest when a participant mentions them. Opening questions slowly guide the conversation towards an area of interest. After understanding what the interviewee's perspective, the researcher can guide the session to provide more detail.

The constructionist endeavors to learn the meanings behind the participant's words or phrases, and not assume that the interviewer and interviewee share a common lexicon, world view, or shared understanding. The constructionist aims to decrease possible preconceptions by understanding the participants' world views.

The interviewer balances the needs of entering into the interviewee's narrative with getting enough detail to understand the emergent theory. Charmaz argues against placing time limits on interviews. Sometimes it is hard for the interviewer to balance listening to the participant while exploring topics of interest. For example, I thought that one of my interviews was wasteful since the interviewee spent 16 minutes answering one single question. Several times during the interviewee's answer, I was tempted to interrupt. During data analysis, however, this segment turned into a treasure trove. The researcher's anxiety might lead to prematurely moving on and missing interesting data.

Sometimes asking a follow-up question such as "could you tell me more or what did you mean by _____" will result in the interviewee provided additional, valuable information.

For organizational studies, Charmaz leads in with questions about collective practices and follows up with questions about the interviewee's participation and point of view. For emotionally charged topics, an interview should proceed with safer questions before diving into personally challenging ones.

When dealing with hard to discuss topics, Charmaz suggests asking the participant " could I ask you about _____?" instead of being direct with a question such as "tell me about _____?" [11]. This question reduces the amount of control the interviewer has in the interview by empowering the interviewee.

When ending an interview, Charmaz prefers the question "is there something?" over a more traditional "is there anything?" The former assumes that there is something that the person should reveal. The later tends to signal that the interview is ending and tends to close the conversation. I tried the variant, "tell me one more thing" and it worked well for me.

Classic grounded theorists argue for note taking during the interview so that the researcher documents the essentials without getting lost in the details [19, 22]. A constructivist finds value in the details.

Constructivist grounded theorists argue that recording the interview enables the interviewer to focus on the narrative without worrying about capturing information during the interview. The transcription and coding process delays concerns about what is relevant and allows researchers to make these decisions at their leisure.

I found that recording the interview, creating a transcript, and coding while listening to the audio helped me identify some of my preconceived ideas. During the coding process, while listening to audio recordings of the interview, I realized that I occasionally misunderstood what the participant said, interpreted what they said in my frame of reference, or ignored something the participant said that I could have followed up on. Reviewing the interview audio enabled me to become a better interviewer as I now had a feedback loop. Since classic grounded theorists do not record the interview, this feedback loop is unavailable to them.

As the research progresses, the interview questions change and respond to the research direction emerging from Grounded Theory. This allows the researcher to direct the data collection and theoretical ideas incrementally. To summarize, Charmaz describes an "intensive interview" process. The process involves open-ended questions. The purpose is for the researcher to enter into the participant's personal perspective within the context of the research question. The interviewer needs to abandon assumptions and personal presumptive to understand and explore that of the interviewee.

Interview Guides: An interview guide is an ordered list of sample questions that the interviewer may use to guide the interview. Interviewers should not see the guides as scripts. The interviewer needs to be present with the interviewee, listening to both verbal and non-verbal communication. The interview guide facilitates a semi-structured interview where the interview asks open ended questions designed to initiate a discussion and follow-up with appropriate questions that emerge from the discussion.

Charmaz advocates that novices rely on interview guides. Creating an interview guide causes the researcher to reflect on sequencing and building of questions, and it reduces the chance of preconceiving the data through leading questions. First, the interview guide serves as a forcing function for the researcher to consider what data they want to collect and the best questions to guide the participant towards those topics. The writing of a guide enables the researcher to consider both how and when to ask difficult questions. Second, relying on an interview guide will decrease the chances of a novice researcher blurting out an inappropriate question, in a moment of panic, or from accidentally asking a leading question. A leading question can bias the interviewee towards a particular point of view by how the question is framed. Grounded Theory emphasizes the emergence of information, not forcing the data through preconceived ideas.

Experts tend to internalize the interview guide and follow where the conversion is leading them. The interview guide can help researchers remove their preconceived perspective which might taint questions. Reflecting on the interview guide helps the researcher accomplish their research objectives.

While grounded theorists agree that researchers should not preconceive the data or the analysis, grounded theorists disagree on what techniques constitute "forcing" the data. Glasser [21] argues against rules for proper memoing, interview guides, and units for data collection, whereas Charmaz argues that an open-ended interview guide for a semi-structured interview would help prevent novices from accidentally asking loaded questions.

Charmaz crafts questions to elicit certain kinds of information. For example, "as you look back on your illness, which events stand out in your mind?" to have participants discuss time sequencing. Glaser might argue that this is forcing the data. Charmaz believes that it opens up commonplace aspects of life.

4.1.1 Interviewing Challenges

There are potential barriers such as power imbalances, social norms, and prior knowledge, and forcing the data, that may affect the interview process and may affect what participants reveal.

The relationship of the researcher and the participant, called "identity" by Charmaz, may affect the interview process. For example, power imbalances of a manager and employee or a professor and students. Different kinds of strategies can be employed. A researcher could distance oneself from positions of authority: a professor interested in learning techniques may decide to interview students in a different degree program. To alter power imbalance, a domain expert researcher might offer "personal and professional views to encourage reciprocity" [11].

Social norms and customs, called "etiquette" by Charmaz. Participants might not want to reveal information to a stranger. A company rule of secrecy might make it difficult for a researcher to get the needed information. Framing a question is one way to overcoming this challenge. For instance, asking the question, "some people have mentioned having negative pair programming sessions; has that happened to you?" normalizes a negative experience, potentially giving permission to the interviewee to discuss an uncomfortable situation.

Prior knowledge can aid or create difficulties for the researcher. A researcher with domain expertise can know about interesting research questions and how to acquire certain kinds of data. That same domain expertise could blind the researcher to possible explanations that an outsider might see.

Preconceived ideas can be used as starting points for open-ended research. A preconceived idea may inspire a researcher to select an area of passion as long as the data guide them. The key is not to force preconceived ideas onto the data. The researcher can initiate with an interesting idea, yet it is critical for the researcher to be flexible with the research topic, listen to the ideas of the participants, and discover the emergent theory.

Inaccurate interviews could occur as it is possible for the participant to accidentally or intentionally fabricate their experience. In this case, the researcher may be entering an implicit collusion with their participants [77]. The constant comparison phase should reveal that the interview is an outlier from the other participants. Instead of treating such interviews as valueless, Grounded Theorists would examine the interchange trying to understand what is happening which may reveal information about forbidden topics and vulnerabilities of both parties. The after-the-fact analysis may help the researcher from repeating such a performance by understanding how the participant was redirecting the interview.

4.2 Field Notes for Participant Observation

The researcher can record observations as field notes, either as a participant involved in the activity. Recording field notes is complementary to interviewing.

Field notes might

- "Record individual and collective actions
- Contain full, detailed notes with anecdotes and observations
- Emphasize significant processes occurring in the setting
- Address what participants define as interesting or problematic
- Place actors and actions in scenes and contexts
- Become progressively focused on key analytic ideas." [11]

Grounded Theory helps with participant observation studies. "Grounded theorists select the scenes they observe and direct their gaze within them. Their field-notes show the actions, processes, and events that constitute what is happening in the setting. Grounded Theory methods provide systematic guidelines for probing beneath the surface and digging into the scene." [11].

Collecting and analyzing field notes from participant observation is not easy. At the beginning of her research, Charmaz pictured a research experience where she would observe her research participants during the day and disappear into the coffee room to take detailed notes and write up her experience [11]. She discovered that this is challenging in practice, as it can be difficult to disappear while observing activities. When the participant activity is intensive, such as the case with pair-programming, recording observations is tricky. I relied on small notes on post-it notes (which were culturally acceptable compared to typing on a laptop) and detailed observations after work.

4.3 Documents and other sources of data

Charmaz suggests that researchers can undervalue documents and encourages researchers to view them as text assets that researchers can mine in the same way as interview notes or field observations. Since the researchers didn't create the document, some might see them as more "objective."

Charmaz identifies two categories of documents, extant documents which exist prior to the researcher's involvement (examples include stories, drawings, code, retro topics, corporate wiki pages, and contracts), and elicited documents which the researcher creates (the researcher interviews participants to capture an account of their experiences). With extant documents, the researcher needs to understand the context in which the document is situated. The researcher can use both types of documents to satisfy possible research goals.

Lindsay Prior [48] argues that more than another voice, documents can do things that a participant can not. Beyond "what do they contain?" the researcher can ask the questions, "what does the document do?", "why was it created?", "how was it created?", "how are the documents used?", "how do people interpret the document?" and "what is not included in the document?"

4.4 Coding

Coding is the process of taking new data and systematically labeling portions of the data that are relevant to the research study. For example, this portion of the transcript, *"Sometimes I kind of feel like a janitor to it. Maybe caretaker would be better. Yeah, probably caretaker like I feel like this janitor just cleans up messes but a caretaker like also makes things better."* could be labeled with "caretaking the code by making it better," "cleaning up messes," and "dealing with technical debt."

The purpose of coding is to fracture the source data into pieces then compare the data against itself during the constant comparison activity. Coding begins the framing from which the researcher builds analysis. In Constructivist Grounded Theory, interviews are recorded, transcribed, and then coded.

Charmaz advises to code everything during the early stages of research and see what emerges. During the initial phase, the researcher should be open to any direction. Once emergent themes arrive, then coding in later parts of the research can be focused around the themes. The focused coding phase allows the researcher to "sort, synthesize, integrate, and organize large amounts of data" [11].

While, there is no perfect coding scheme, there are different styles of coding [54]. Starting with Glasser in 1978 [19] some grounded theorists argue for using gerunds (e.g. “dealing with uncertainty,” “exploring solutions to a problem”) instead of topic or noun-based coding schemes (e.g. “uncertainty,” “solutions”). Relying on gerunds helps encourage the researcher to dig into the data and see the relationship between the participant and their actions. Charmaz strongly argues against labeling events, experiences, or topics as codes as the researcher gains little insight into the participant’s meaning, action, or point of view. Charmaz prefers a coding style that is simple, short, direct, advice, analytic and spontaneous. Both the researcher’s and the participants’ vocabulary influences the coding. Because Strauss and Corbin [64] place less emphasis on this distinction, many grounded theorists take a more open stance to coding than relying on gerunds. Both Glaser and Charmaz do not want the researcher to bring preconceived ideas to coding.

There are different granularity of coding. Charmaz prefers line-by-line coding and suggests there are times when even word-by-word coding is helpful. The line-by-line coding helps the researcher slow down and examine for nuanced interactions in the data. By 1992, Glasser prefers topic-by-topic or incident-to-incident coding. He advises against decomposing a single incident. He feels that line-by-line coding produces too many codes, categories, and properties without necessarily supporting the analysis. While line-by-line coding does generate more codes, I appreciated the intimacy and thoroughness of line-by-line coding.

Charmaz suggests these heuristics to guide the researcher in the coding process:

- “Remain open
- Stay close to the data
- Keep your codes simple and precise
- Construct short codes
- Preserve actions
- Compare data with data
- Move quickly through the data” [11]

Charmaz suggests these strategies when examining source materials:

- “Breaking the data up into their component parts or properties
- Defining the actions on which they rest
- Looking for tacit assumptions

- Explicating implicit actions and meanings
- Crystallizing the significance of the points
- Identifying gaps in the data" [11]

Whenever an insight arises or a "instantaneous realization of analytic connections" occurs, the researcher immediately stops the coding process and records the idea as a memo.

For the constructivist, the researcher creates the code when the researcher examines the data and finds meaning within it. "We *construct* our codes because we are actively naming data . . . We choose the words that constitute our codes. Thus we define what we see as significant in the data and describe what we think is happening" [11].

Sometimes a researcher sees everything as trivial or everything as significant. This is normal and in time, the data collection and coding process will sort out the core concerns of the participants. When observing routine activity, it can be challenging to see anything meaningful in the data. In this situation, the researcher can compare events to events looking for similarities and differences.

If for some reason the codes remain mundane, Charmaz suggests "coding the codes." In examining the codes, there may be a larger process or activity. There might be patterns in the codes.

Tensions may emerge in the coding. The researcher should embrace tensions rather than avoid or hide them. Tensions may become properties of categories, differentiating key aspects of the data.

At the researcher codes the data, Charmaz encourages the researcher to answer these questions:

- "What process(es) is at issue here? How can I define it?
- How does this process develop?
- How does the research participant(s) act while involved in this process?
- What does the research participant(s) profess to think and feel while involved in this process? What might his or her observed behavior indicate?
- When, why, and how does the process change?
- What are the consequences of the process?" [11]

"Grounded theorists aim to code for possibilities suggested *by* the data rather than ensuring complete accuracy *of* the data" [11]. This stance provides opportunities for checking envisioned ideas with other data. Constant comparison allows the researcher to invalidate conjecture; an errant code will be detected and unsupported by other data samples. Ideas reflected in the data must earn their way into subsequent analysis. Constructivist grounded theorists acknowledge that both the researcher's and the participants' vocabulary influences the coding.

There are two phases of coding: initial coding and focused coding.

The initial coding is the process of coding the first set of data. The initial coding names every part of the data. Initial coding is an intimate process for the researcher as the researcher becomes familiar with the data. The researcher can examine the data looking for implied meanings. The emphasis is on summarizing the data and not yet analyzing it. The researcher tries to avoid asking what the data means at this stage. Both actions and processes can be coded. After starting initial coding, the researcher begins using constant comparison described in the next section. Constant comparison helps shape the direction of the research.

Focused coding is the second phase of coding. Once core categories emerge from constant comparison, the researcher shifts from initial coding to focused coding where the emphasis is on flushing out the core categories and their properties. While it is hard to predict when the transition will occur, Charmaz sees a natural transition when the researcher simply starts performing focused coding. Focused coding continues until theoretical saturation occurs, as described in the section on theoretical sampling.

As a reminder, the data advances from the initial codes to focused codes, focused codes to core categories, and core categories to an emergent theory.

4.5 Constant Comparison

Constant comparison allows the researcher to identify "the conceptual relationship between categories and their properties as they emerged" [20], leading to an emergent theory.

In constant comparison, the researcher examines and compares codes to codes. It may be the case that two codes should be combined since they are describing the same phenomena. Codes that are related form a category. The researcher compares category to category looking for the relationship between them. The researcher periodically audits each category for cohesion by comparing its codes and moves codes that belong to a different category. Constant comparison compares incident to incident looking for patterns. Similarities strengthen the category while differences refine properties of the category.

As categories emerge from the data, the researcher is looking for the core category as it defines the chief concern of the participants. Once the core category emerges, the researcher continues to collect and analyze data by examining the properties of the core category and its relationship to other categories until theoretical saturation occurs.

The researcher pauses to record memos for any insights that occur during the analysis.

4.6 Memo-writing

Memo-writing is writing down the researcher's insights and analysis. Memoing can occur at any point in the process, and preempts any other activity. "Memo-writing is the pivotal intermediate step between data collection and writing drafts of papers" [11].

Memo-writing pauses the other research activities and allows the researcher to reflect on what is going on in the data and analysis. Memo-writing encourages the researcher to analyze the data early in the process. The output of memo-writing can enable the researcher to adjust the course of the research process.

Memos written early in the research process may be more tentative and less theoretically developed than memos written later in the research process.

Charmaz encourages the researcher to "engage an emergent category, let your mind rove freely in, around, under, and from the category; and write whatever comes to you. That is why memo-writing forms an interactive space and place for exploration and discovery. You take the time to discover your ideas about what you have seen, heard, sensed, and coded and then you examine these ideas" [11].

Some researchers use a "Methodological Journal" in which the researcher records dilemma, directions, and decisions. A journal helps the researcher reflect about the data and the process. The key is for the researcher to find a system that works for them.

For early memos, Charmaz recommends that researchers "record what they see happening in the data" and to use early memos to explore and fill out qualitative codes [11].

For memos later in the process, Charmaz suggests to

- "Trace and categorize data subsumed by the topic.
- Describe how the category emerges and changes.
- Identify the beliefs and assumptions that support it.
- Specify how the category informs action and experience.
- If relevant, tell what it looks and feels like from various vantage points.
- Place the category or categories within an argument.
- Sharpen comparisons of people, data, codes, categories, subcategories, concepts, and analysis with existing literature." [11]

There is no standard, correct memo. Contents vary from memo to memo. Memos can have a title. Memos tend to be private, unshared documents enabling the researcher to be perfectly free with thoughts, not worrying about grammar, editing, and future readers. The paper writing process will sort all that out. As such, researchers can record doubts, concerns, and first thoughts. Charmaz lists several ways researchers can use a memo:

- "Define each code or category by its analytic properties
- Spell out and detail processes subsumed by the codes or categories
- Make comparisons between data and data, data and codes, codes and codes, codes and categories, categories and categories
- Bring raw data into the memo
- Provide sufficient empirical evidence to support your definitions of the category and analytic claims about it
- Offer conjectures to check in the field setting(s)
- Sort and order codes and categories
- Identify gaps in the analysis
- Interrogate a code or category by asking questions of it." [11]

In order to facilitate memo-writing, Charmaz suggests relying on several writing exercises to stimulate the writing process such as freewriting (keep writing whatever comes to the mind) and clustering (diagramming relationships).

Memo-writing can raise focused codes to conceptual categories. As the researcher does the analysis on focused codes, the researcher identifies which codes describe what is occurring in the data. A category can span several codes.

Both Glaser and Charmaz desire conceptual categories with "abstract power, general reach, analytic direction, and precise wording" [11] while grounded in the data. Conceptual categories may be applicable in different domains, professions, and fields and help explain substantive processes. Examples include "getting off the street" [32], "managing *spoiled* national identity" [53], and "living one day at a time". Focusing on generic processing raises codes to theoretical categories.

As the key analytic step, memos become the foundation of the emergent theory. The researcher may deem some memos as complete whereas others raise unanswered questions. Revisiting previous memos allows the researcher to see what is missing in the memo and directs new research directions. Memo-writing helps the researcher to identify additional data to collect.

4.7 Theoretical Sampling

Theoretical sampling is collecting additional data to develop full and robust categories, identify the relationships between categories, and flush out the main category's properties.

For example, data and codes may yield unanswered questions and categories may not be definitive and may suggest new avenues of exploration. Additional data collection, coding, and analysis refine the emergent theory which produces a new vantage point for further exploration and refinement. Memo-writing encourages theoretical sampling as the researcher reflects on the data and the analysis causes new questions to emerge.

Activities include adding new participants, observing in different settings, re-interviewing participants with follow-up questions or ask about different kinds of experiences.

This process continues until the category structure stabilizes and nothing new can be added to the theory, i.e. theoretical saturation.

Theoretical sampling is not

- "Sampling to address initial research questions
- Sampling to reflect population distributions
- Sampling to find negative cases
- Sampling until no new data emerge." [11]

There's a subtle distinction between "sampling until no new data emerges" and sampling to flesh out the emerging theory. In discussing theoretical sampling or "saturation" with software engineering researchers practicing Grounded Theory, I misconstrued that theoretical sampling is the continued collection of data until no new data and thus no new insights emerge. Stopping makes sense if the collection of data results in the same kind of information. However, theoretical sampling has a subtle distinction from this misconception. Grounded Theory is not about asking the same questions over and over again until the same questions result in no new information. Grounded Theory alters the questions in response to the emerging data, and the researcher continues to ask new questions to help fill in the emerging theory. Thus, theoretical sampling is collecting new information to illicit the relationship between codes, between codes and categories, and between categories. The process stops once the researcher has matured the theory, and there is nothing more to ask the participants.

In review, theoretical sampling is collecting additional data to develop full and robust categories, identify the relationships between categories, and flush out the core category's properties.

4.7.1 How much data to collect?

The researcher continues to gather data until theoretical saturation occurs. At the beginning of a research study, the researcher can not predict when saturation will happen. Thus, there are no useful metrics for how much data to collect. Likewise, it is challenging for a reviewer to determine if a study has collected enough data. Clearly, more data is better than less data. Charmaz points out asking how many interviews is enough is asking the wrong question. She would prefer to have enough interviews that strengthen the research with deep and sufficient vigor.

Studies using mixed research methods may require fewer interviews, as the study relies on data from different data sources.

Data should give a full picture. Two key aspects of data for depicting empirical events are suitability and sufficiency. The researcher needs to plan to gather sufficient data.

For Glasser [21] and Stern [62], small data samples, and limited data is not an issue since the purpose of Grounded Theory is to create conceptual categories. Charmaz argues that limited data can lead to weak analysis.

4.8 Memo Sorting

Memo sorting involves the sorting, comparing, and integrating of the memos to refine the emergent theory. Sorting involves examining different arrangements of the memos to determine which sequence clearly tells the story of the theory. Often memos describing properties of a category are sequenced around the category. If there is a time sequence to a process, memos might be sorted chronologically with the steps of the process. Charmaz suggests printing them out, arranging them on the floor or a dining room table, and rearranging them until a natural progression emerges. Comparing involves juxtaposing two memos and seeing if new insights and thus a new memo emerges. Integrating involves fitting the memos together into a cohesive whole.

Thus memo sorting is more than simply creating the first draft. The process of memo sorting stimulates additional analytical work, may raise new questions that may stimulate additional data collection, coding, and analysis.

The memo sorting phase begins the process of helping the researcher articulate the theory.

4.9 Theory Construction

For researchers who do want to generate theory, Charmaz suggests that the researcher attends to four concerns: theoretical plausibility, direction, centrality, and adequacy. These concerns arise because the researcher can direct and control the data generation, and thus the emerging theory. Although decomposed into four concepts, the key idea of some studies may span several of these concerns at the same time.

While Charmaz does not define these terms in her book, the following definitions are gleaned by the contextual usage of each team.

Theoretical plausibility reflects whether an idea might turn into a theory. Often key interview statements might represent some larger notion. The researcher treats repeated themes as theoretically plausible.

Theoretical direction means that the data and the analysis are pointing the researcher towards a particular path. After the initial interviews and the initial coding, a theoretical direction emerges from the data. Certain ideas and codes routinely emerge from the data. The data suggests paths that need exploring. Without picking a direction based on the emergent data, the researcher could languish in the early stages of research without iterating towards an emergent theory.

Theoretical centrality is the researcher defining what is core to the research. Once the researcher has explored these paths, theoretical centrality emerges from the coding and analysis. Certain codes become core to the research. The researcher drops less fruitful paths and codes as the researcher focuses the interviews on the central theme or categories.

Theoretical adequacy is the assessment of the robustness of the emergent theory in its categories and properties as verified in theoretical sampling. During later interviews, the researcher introduces questions for theoretical sampling and to gauge the robustness of the emergent categories. Theoretical adequacy is central to theoretical sampling and saturation.

For the constructivist, theoretical plausibility supersedes interviewee accuracy. Charmaz reminds that definitions of accuracy are social constructs. Charmaz argues that the amount of data collected in a Grounded Theory study typically offsets any adverse effects of "misleading accounts" and thus decreases the probability that the research's work would have spurious results. "Grounded Theory aims to make patterns visible and understandable" [11]. Thus a grounded theorist should strive to have wide and deep coverage of their categories.

If a participant does offer exaggerated or inaccurate accounts, and the researcher detects this situation, then this can be a research opportunity into understanding how the candidate is creating fictional representations of their situation. Charmaz recounts seasoned citizens retaining identity patterns from earlier in their life; for example, one person described how she daily attended her garden even though she had not done so for years.

Glaser's chief goal is in theory building and routinely emphasizes conceptualization. Theoretical coding requires the researcher to think about the relationship between the codes. Coding families, groups of similar codings, help the researcher consider possible relationships between codes. In his book, Theoretical Sensitivity [19], Glaser lists 18 theoretical coding families and acknowledges that there may be more:

1. "*The Six C's:* Causes, Contexts, Contingencies, Consequences, Covariances and Conditions

2. *Process:* Stages, staging, phases, phasings, progressions, passages, gradations, transitions, steps, ranks, careers, orderings, trajectories, chains, sequencings, temporaling, shaping and cycling.

3. *The Degree Family:* Limit, range, intensity, extent, amount, polarity, extreme, boundary, rank, grades, continuum, probability, possibility, level, cutting points, critical juncture, statistical average (mean, medium, mode), deviation, standard deviation, exemplar, modicum, full, partial, almost, half and so forth.

4. *The Dimension Family:* Dimensions, elements, division, piece of, properties of, facet, slice, sector, portion, segment, part, aspect, section. The dimension family divides the notion of a whole into a parts.

5. *Type Family:* Type, form, kinds, styles, classes, genre. While dimensions divide up the whole, types indicate a variation in the whole, based on a combination of categories.

6. *The Strategy Family:* Strategies, tactics, mechanisms, managed, way, manipulation, maneuverings, dealing with, handling, techniques, ploys, means, goals, arrangements, dominating, positioning.

7. *Interactive Family:* Mutual effects, reciprocity, mutual trajectory, mutual dependency, interdependence, interaction of effects, covariance.

8. *Identity-Self Family:* Self-image, self-concept, self-worth, self-evaluation, identity, social worth, self-realization, transformations of self, conversions of identity.

9. *Cutting Point Family:* Boundary, critical juncture, cutting point, turning point, breaking point, benchmark, division, cleavage, scales, in-out, intra-extra, tolerance levels, dichotomy, trichotomy, polychotomy, deviance and point of no return.

10. *Means-goal Family:* End, purpose, goal, anticipated consequence, products.

11. *Cultural Family:* Social norms, social values, social beliefs, and social sentiments.

12. *Consensus Family:* Clusters, agreements, contracts, definitions of the situation, uniformities, opinions, conflict, dissensus, differential perception, cooperation, homogeneity-heterogeneity, conformity, nonconformity, and mutual expectation.

13. *The Mainline Family:* Social control (keeping people in line), Recruitment (getting people in), Socialization (training people for participation), Stratification (sorting people out on criteria which rank them), Status Passage (moving people along and getting them through), Social organization (organizing the people into groups, aggregates and divisions of labor) and Social Order, (keeping the organization of life working normatively), Social institutions (clusters of cultural ideas), Social interaction, (people acting with people), Social worlds (symbolic surround of life), Social mobility (patterned paths of people movement through society) and so forth.

14. *Theoretical Family:* Parsimony, scope, integration, density, conceptual level, relationship to data, relationship to other theory, clarity, fit, relevance, modifiability, utility, condensability, inductive-deductive balance and inter-feeding, degree of, multivariate structure, use of theoretical codes, interpretive, explanatory and predictive power, and so forth.

15. *Ordering or Elaboration Family:* Structural, temporal and generality are the three principal ways to order data.

16. *Unit Family:* Collective, group, nation, organization, aggregate, situation, context, arena, social world, behavioral pattern, territorial units, society, family, etc., and positional units: status, role, role relationship, status-set, role-set, person-set, role partners.

17. *Reading Family:* Concepts, problems, and hypotheses.

18. *Models:* Another way to theoretically code is to model the theory pictorially by either a linear model or a property space." [19]

4.10 Evaluation

Glaser provides the criteria of fit, work, relevance, and modifiability for determining how well an emergent theory explains the data [19]. Stol et al. [63] explain these criteria as:

- **Fit**: "The generated categories must fit the data"
- **Work**: "It must be able to explain or predict what will happen"

- **Relevance**: "The theory must have relevance to the action of the area"
- **Modifiability**: "The theory must be modifiable as new data appear"

Charmaz provides the criteria of credibility, originality, resonance, and usefulness. Stol et al. [63] summarizes these criteria as:

- **Credibility**: "Is there sufficient data to merit claims?"
- **Originality**: "Do the categories offer new insights?"
- **Resonance**: "Does the theory make sense to participants?"
- **Usefulness**: "Does the theory offer useful interpretations?"

4.11 Tool Support

Some qualitative researchers prefer using qualitative research tools such as NVivo and Atlas.ti for the process of coding transcriptions and facilitating analysis of codes. Many grounded theory researchers prefer using word processing or spreadsheet software. Before computers, Glaser recommended using carbon copy paper so that one copy of the transcripts and memos could be cut and rearranged for analysis while the other copy preserves the original data. The researcher needs to select the tools that facilitate their workflow and provide easy access to the data they need at each step of the Grounded Theory research process.

While NVivo 11.0.0 for Mac supports adding codes to a transcript, there is no keyboard shortcut to add a code, and there is no auto-complete for existing codes in the system. The researcher needs to remember all the codes already added to the system. The cognitive overhead of remembering when to add a new code or re-use an existing code could be too much. The tool might be good for the focused coding with a fixed set of codes. The interface for peer reviewing codes was extremely challenging to use.

Atlas.ti 1.0.24 has keyboard shortcuts and auto-complete suggestions for existing codes thus creating a better initial coding workflow. However, there is no way to edit the source material. While coding a transcript, if the researcher notices a mistake in the transcription, there is no way to fix it.

Microsoft Word is simple to use for transcriptions and initial coding with a three column format. The first column contains a unique id reference for that row for the purpose of cross-referencing during constant comparison. The second column is for initial coding and focused coding. The third column is for the transcribed interview, breaking sections into new rows. Recording timestamps for each segment allows for cross referencing with the original audio.

The researcher can export the word document into a comma separated file with the id and coding columns. The researcher then imports the file into a spreadsheet for constant comparison and analysis. Sometimes constant comparison can be easy to do in the spreadsheet. Sometimes it is easy to print index cards for physically sorting the cards around the researcher. The researcher can use mail merge to print specific rows into a Microsoft Word template for Avery shipping labels. After printing the labels, the researcher applies the sticker to index cards. If desired, colored index cards can be used to distinguish each data source. After sorting and clustering, the researcher would need to update the spreadsheet to match the physical arrangement of cards.

Any system could work for a researcher provided that the system facilitates coding and constant comparison.

4.12 Constructivism

The Constructivist approach to Grounded Theory "emphasizes understanding and acknowledges that data, interpretations, and resulting theory depend on the researcher's view" [63].

The constructivist acknowledges that the "humanness" of the researcher may affect the research. Constructivists attempt to identify their assumptions and not accidentally reproduce the assumptions in the research results. Relying on and listening to the participants helps counteract researcher bias.

The interplay between the researcher and the participant during an interview is particularly interesting from a constructivist perspective. Interviews are situated within a particular context. Changes in location, time, and setting would result in different data. Charmaz argues that interviews are a construction between the interviewee and the interviewer. "The result is a construction - or reconstruction - of reality. Through constructing their respective performances, interviewers and interview participants present themselves to each other. However silent, both the interviewer's and participant's performances make and negotiate identity claims" [11].

Because Charmaz has a constructivist perspective, she acknowledges that "neither observer nor observed come to a scene untouched by the world" [11]. The research method affects the kind of data observed, the researcher's background affects the observations that the researcher can see. The onus is on the researchers to bring scrutiny and reflective practice to understand how their point of view may bias observations. Charmaz points out that "*people* construct data" through field notes and interviews. While it is tempting to treat an extent report or document as fact, constructivists remind themselves that people collected and formed them.

The contrasting positions of constructivism and positivism made me wonder about my personal stance and how it might affect the research process. While I believe that there are absolute facts to be obtained in fields such as mathematics and the sciences, understanding people is a messy, organic process. For software engineering research, Constructivist Grounded Theory seems aptly suited since software development is a socio-technical endeavor.

User research is a pragmatic process attempting to identify easy-to-find insights, not generate an exhaustive theory, and thus fewer interviews seem necessary.

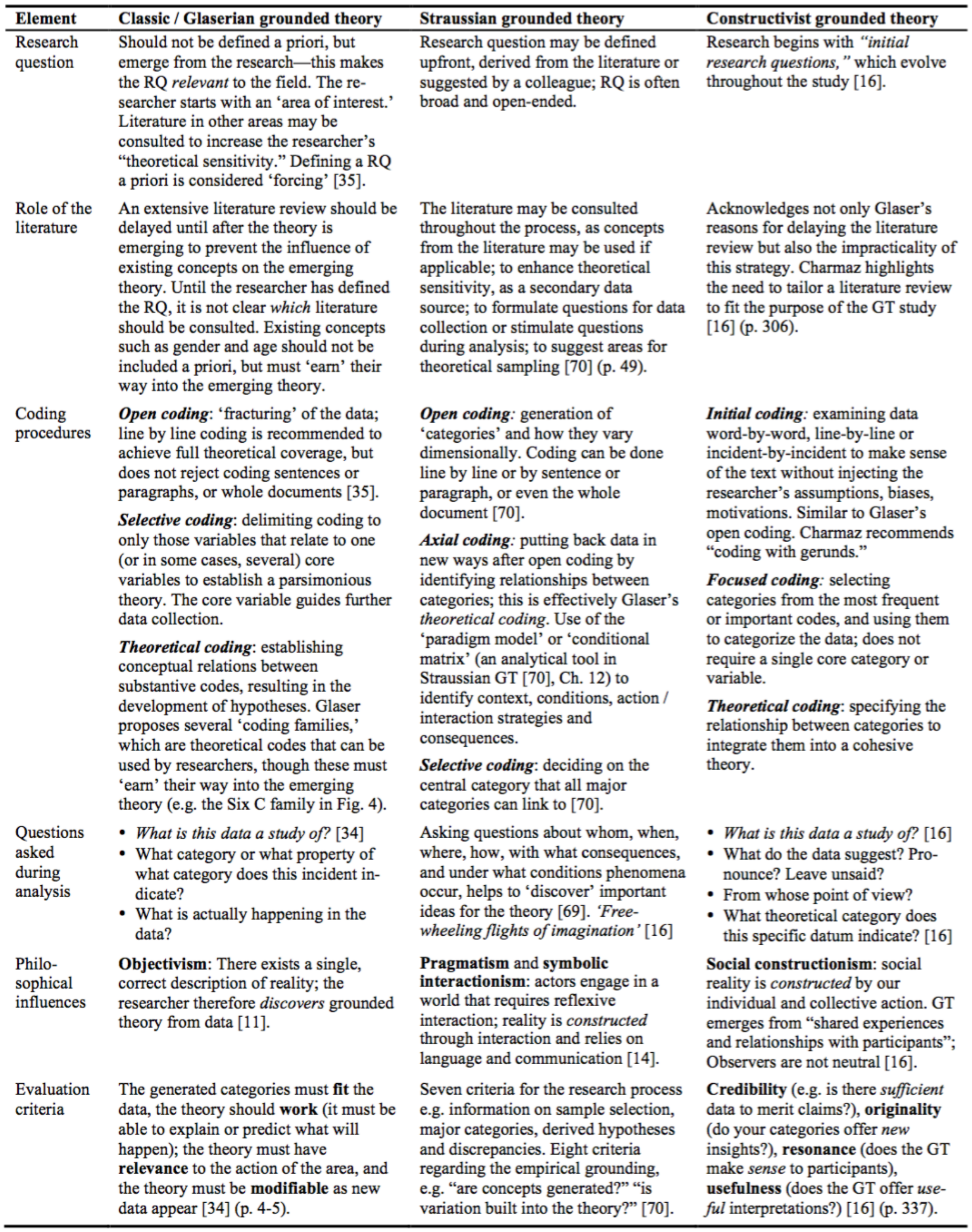
4.13 Conclusion

Starting a Grounded Theory research study, the researcher asks the core question “what is happening here?” [19]

Grounded Theory allows the researcher to include new aspects of the research while gathering data, even late in the analysis. The data collection process is guided by the research and altered as knowledge is accumulated. It is common for early research to illuminate new angles. “Grounded Theory can give you flexible guidelines rather than rigid prescriptions” [11]. The theory provides a starting place for inquiry, not a specific goal known at the beginning of the research. The researcher’s guiding interests can be used as points of “departure to form interview questions, to look at data, to listen to interviewees, and to think analytically about the data” [11].

After interviewing, data analysis begins with line-by-line coding as recommended by Charmaz [11]. 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.

Table 4.1: Stol's Grounded Theory Comparison



Element	Classic / Glaserian grounded theory	Straussian grounded theory	Constructivist grounded theory
Research question	Should not be defined a priori, but emerge from the research—this makes the RQ *relevant* to the field. The researcher starts with an 'area of interest.' Literature in other areas may be consulted to increase the researcher's "theoretical sensitivity." Defining a RQ a priori is considered 'forcing' [35].	Research question may be defined upfront, derived from the literature or suggested by a colleague; RQ is often broad and open-ended.	Research begins with *"initial research questions,"* which evolve throughout the study [16].
Role of the literature	An extensive literature review should be delayed until after the theory is emerging to prevent the influence of existing concepts on the emerging theory. Until the researcher has defined the RQ, it is not clear *which* literature should be consulted. Existing concepts such as gender and age should not be included a priori, but must 'earn' their way into the emerging theory.	The literature may be consulted throughout the process, as concepts from the literature may be used if applicable; to enhance theoretical sensitivity, as a secondary data source; to formulate questions for data collection or stimulate questions during analysis; to suggest areas for theoretical sampling [70] (p. 49).	Acknowledges not only Glaser's reasons for delaying the literature review but also the impracticality of this strategy. Charmaz highlights the need to tailor a literature review to fit the purpose of the GT study [16] (p. 306).
Coding procedures	***Open coding***: 'fracturing' of the data; line by line coding is recommended to achieve full theoretical coverage, but does not reject coding sentences or paragraphs, or whole documents [35]. ***Selective coding***: delimiting coding to only those variables that relate to one (or in some cases, several) core variables to establish a parsimonious theory. The core variable guides further data collection. ***Theoretical coding***: establishing conceptual relations between substantive codes, resulting in the development of hypotheses. Glaser proposes several 'coding families,' which are theoretical codes that can be used by researchers, though these must 'earn' their way into the emerging theory (e.g. the Six C family in Fig. 4).	***Open coding****:* generation of 'categories' and how they vary dimensionally. Coding can be done line by line or by sentence or paragraph, or even the whole document [70]. ***Axial coding****:* putting back data in new ways after open coding by identifying relationships between categories; this is effectively Glaser's *theoretical coding*. Use of the 'paradigm model' or 'conditional matrix' (an analytical tool in Straussian GT [70], Ch. 12) to identify context, conditions, action / interaction strategies and consequences. ***Selective coding***: deciding on the central category that all major categories can link to [70].	***Initial coding****:* examining data word-by-word, line-by-line or incident-by-incident to make sense of the text without injecting the researcher's assumptions, biases, motivations. Similar to Glaser's open coding. Charmaz recommends "coding with gerunds." ***Focused coding****:* selecting categories from the most frequent or important codes, and using them to categorize the data; does not require a single core category or variable. ***Theoretical coding***: specifying the relationship between categories to integrate them into a cohesive theory.
Questions asked during analysis	• *What is this data a study of?* [34] • What category or what property of what category does this incident indicate? • What is actually happening in the data?	Asking questions about whom, when, where, how, with what consequences, and under what conditions phenomena occur, helps to 'discover' important ideas for the theory [69]. *'Free-wheeling flights of imagination'* [16]	• *What is this data a study of?* [16] • What do the data suggest? Pronounce? Leave unsaid? • From whose point of view? • What theoretical category does this specific datum indicate? [16]
Philosophical influences	**Objectivism**: There exists a single, correct description of reality; the researcher therefore *discovers* grounded theory from data [11].	**Pragmatism** and **symbolic interactionism**: actors engage in a world that requires reflexive interaction; reality is *constructed* through interaction and relies on language and communication [14].	**Social constructionism**: social reality is *constructed* by our individual and collective action. GT emerges from "shared experiences and relationships with participants"; Observers are not neutral [16].
Evaluation criteria	The generated categories must **fit** the data, the theory should **work** (it must be able to explain or predict what will happen); the theory must have **relevance** to the action of the area, and the theory must be **modifiable** as new data appear [34] (p. 4-5).	Seven criteria for the research process e.g. information on sample selection, major categories, derived hypotheses and discrepancies. Eight criteria regarding the empirical grounding, e.g. "are concepts generated?" "is variation built into the theory?" [70].	**Credibility** (e.g. is there *sufficient* data to merit claims?), **originality** (do your categories offer *new* insights?), **resonance** (does the GT make *sense* to participants), **usefulness** (does the GT offer *useful* interpretations?) [16] (p. 337).

Chapter 5

Research Method

5.1 Constructivist Grounded Theory

When deciding which version of Grounded Theory to employ, we selected Constructivist Grounded Theory [11] because it allows for the recording and transcription of interviews, enables literature review when appropriate in the research process, and builds upon decades of grounded theory research and experience.

The philosophical stance of constructivism (as opposed to positivism) acknowledges the researcher's and participant's contribution in the construction of concepts. While there are absolute facts to be obtained in fields such as mathematics and the sciences, understanding people is a messy, organic process. For software engineering research, Constructivist Grounded Theory seems aptly suited since software development is a socio-technical endeavor.

As described in Chapter 4, Constructivist Grounded Theory provides an iterative approach to data collection, data coding, and analysis resulting in an emergent theory. Grounded Theory research begins by asking, "what is happening here?" [19]; or in this case, "what is happening at Pivotal when it comes to software development?"

In time, "team code ownership" and "removing waste" emerged as two of the core categories. Exploring theoretical saturation for team code ownership resulted in the theory of sustainable software development presented in Chapter 6 and five factors that affect a team's sense of code ownership presented in Chapter 7. Exploring theoretical saturation for removing waste resulted in a taxonomy of software engineering waste presented in Chapter 8.

5.2 Research Context: Pivotal Labs

Pivotal Labs is a division of Pivotal—a large American software company (with 17 offices around the world). Pivotal Labs provides teams of agile developers, product managers, and interaction designers to other firms. Its mission is not only to deliver highly-crafted software products but also to help transform 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 [8] in an environment conducive to agile development.

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.

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.

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 [8] 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. Only teams at Pivotal Labs were observed. Other teams, especially teams in other divisions, might have a different culture and follow different software practices.

Pivotal is an good research location because: 1) it is successful; 2) it is interesting in its continued use and evolution of extreme programming; 3) it is accessible and cooperative with research. Both Classic and Constructivist Grounded Theory advocate picking an interesting site to see "What's going on here?"

5.3 Data Collection

This research study analyses data from three sources: 1) interviews with Pivotal employees, 2) participant observation of eight projects over two years and five months, and 3) topics discussed in 91 retrospection meetings.

For this grounded theory study, 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.

5.3.1 Interviews

The interviewees consisted of 33 interaction designers, product managers, and software engineers who had experience with Pivotal's software development process from five 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.

We relied on "intensive interviews," which are "open-ended yet directed, shaped yet emergent, and paced yet unrestricted" [11]. 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 [11] contrasts intensive interviews with informational interviews (collecting facts), and investigative interviews (exposing hidden intentions, practices or 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, when team code ownership emerged as one of the core categories, asking the participant, "Please draw your feelings about the code" often resulted in conversations about code ownership. The interviews were spread across the duration of the research study.

5.3.2 Participant Observation

We collected field notes while the lead researcher worked as an engineer on eight projects. These notes describe individual and collective actions, capture what participants found interesting or problematic, and include anecdotes and observations.

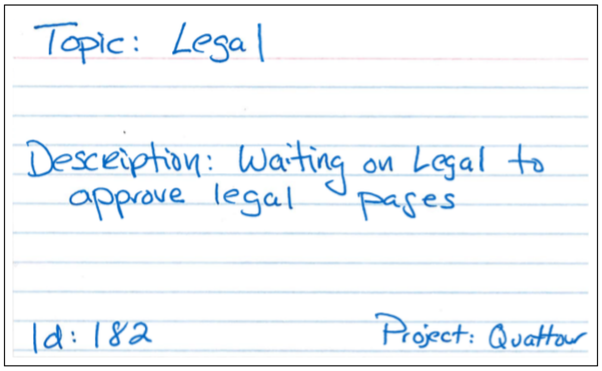
Projects are de-identified to preserve client confidentiality:

- Project Unum (two product managers, four developers) was a greenfield project providing a web front end for installing, configuring, and using a multi-node cluster with big data tools.
- Project Duo (two interaction designers, two product managers, six developers) added features to a print-on-demand e-commerce platform.
- Project Tes (one interaction designer, one product manager, six developers) added features to management software for internet service providers.
- Project Quattuor (two interaction designers, three product managers, 28 developers) developed two mobile applications and a backend system for controlling expensive equipment.
- Project Kvin (one interaction designer, one product manager, six developers) was a greenfield project for a healthcare startup.
- Project Ses (two interaction designers, one product manager, ten developers) was adding features and removing technical debt to an existing internet e-commerce website.
- Project Septem (two interaction designers, three product managers, twelve developers) was adding features and removing technical debt to an existing virtual machine management software.
- Project Octo (one product manager, four developers) added features for workload management of a multi-node database.

5.3.3 Retrospection Topics

When *removing waste* emerged as one of the core categories from interviews and participant observation, we began collecting data from retrospection meetings. A retrospection meeting (or retro) is a meeting to pause, reflect, and discuss the work done during the week, i.e., a safe place where any team member can discuss any issue [15]. Retros are typically scheduled every Friday afternoon. The entire team and important stakeholders attend these meetings.

The observed Pivotal teams mostly use an emotion-based retro format where "happy," "neutral," and "sad" faces are written on the top of a whiteboard. The happy-face column represents items that are working well and should be continued or expanded. The neutral-face column represents items that the team needs to "keep an eye on." The sad-face column represents problems that the team should try to fix. Any team member can add any topic to any column. After a few minutes, the team dot-votes on the topics to discuss [15]. The team uses the remainder of the sixty-minute meeting to discuss topics. Sometimes discussing a topic is sufficient to affect change, other times the team creates action items.


Topic: Legal

Description: Waiting on Legal to approve legal pages

Id: 182 Project: Quattor

Figure 5.1: Example Retro Topic Index Card

We collected data from 91 retrospection meetings over 59 weeks from Projects Quattuor, Kvin, and Ses. (There are more meeting than weeks since each of Project Quattuor's three teams held its own retro each week.)

For co-located teams, a whiteboard picture was taken at the end of the retro and the topics were later transcribed into a master spreadsheet. For distributed teams, we copied data from the online spreadsheets the team used in place of a whiteboard. Attendees often wrote a short phrase as a proxy for a larger idea (e.g., "Scope" represents "Too much scope is causing the team stress"). When the provided topic was too vague, we solicited a more detailed description from an engineer that was present at the meeting. This produced 663 total items for analysis.

5.4 Data Analysis

Data analysis began by iteratively collecting and analyzing interview transcripts and participant observations. We used line-by-line coding [11] to identify nuanced interactions in the data and avoid jumping to conclusions. 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 [19], we recorded and transcribed them into grounded theory memos. As data was collected and coded, we stored initial codes in a spreadsheet and we used constant comparison to generate focused codes.

We routinely 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, we printed codes 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.

When *removing waste* appeared as a core category, we analyzed data from retrospectives to investigate (theoretical sampling). After removing irrelevant topics (e.g. complaints about the weather), we printed each retro item onto an index card with its original retro topic, enhanced description, ID, and team name (see Figure 5.1).

Two researchers with first-hand experience of the projects coded the retro topics and merged duplicate topics. We iteratively reorganized categories, keeping similar items together and dissimilar items apart. Figure 8.1 gives an example classification for the *psychological distress* waste. The figure shows the waste category, its cause categories and properties, and examples of observed retrospective topics illustrating the waste. Appendix A presents the entire chain of evidence for all waste categories.

We often stopped to record new insights. When the categories began to stabilize, we compared each category against the other categories looking for relationships. Once we felt that the categories were stable, we performed a final review of each category to verify that the cards belonged to it.

We continued theoretical sampling for team code ownership and removing waste in additional interviews and participant observations until no further categories were evident, i.e. theoretical saturation.

In summary, the data advanced from the initial codes to focused codes, focused codes to core categories, and core categories to an emergent theory.

Chapter 6

Constructing Products: Sustainable Software Development

6.1 Summary

Context: This chapter examines how software teams achieve development sustainability, even in the face of disruption. 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 is to understand how to develop software effectively, even in the face of team disruption.

Method: Following Constructivist Grounded Theory, we conducted a two-year five-month participant-observation of eight projects at Pivotal (a software development company), and interviewed 33 software engineers, interaction designers, and product managers. Iterating between theoretical sampling and analysis continued until achieving theoretical saturation.

Results: This chapter 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 the understanding of Extreme Programming by adding new principles, policies, and practices (including Overlapping Pair Rotation) and aligning them with the business goal of sustainability.

6.2 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 [35], 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.

Section 6.3 describes related work on Extreme Programming and team disruption. Section 6.4, reviews the research method to derive a descriptive theory supported by empirical data. Section 6.4 also presents the research context, introducing both the company and one of the eight projects under study. Section 6.5 describes the theory and how its principles, policies, and practices work together to achieve software development sustainability. Section 6.6 evaluates the theory using established criteria for evaluating a Grounded Theory. The last sections of the chapter examine threats to research validity and summarize the research.

6.3 Related Work

In Extreme Programming [8], Kent Beck describes a set of interdependent practices that manage feature development (much like Scrum [55]), as well as technical practices that facilitate a collaborative team environment. Extreme Programming comprises 13 primary practices and 11 corollary practices as further described in Chapter 2.

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 [7]. Later, “collective ownership” was renamed to “shared code” [8].

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 [8]. 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” [75]. 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 [70]. Some teams use a pair programing matrix [1] (also called a pairing ladder [14]) 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.” [43]. Truck Number, or Bus Count, reminds management about the effects of disruptive events for a team. In 1994, Coplien [13] 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, increasing cross-training, and increasing documentation [5], all of which are found in Extreme Programming. However, Extreme Programming implements “documentation” as discoverable, intention revealing code. Ricca [51] examines the difficulty in computing the Truck Number.

Rigby quantifies turnover using knowledge at risk analysis on abandoned files [52]. 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 [28]. Joseph examines job turnovers to understand the reasons developers leave their company [30].

6.4 Research Method

6.4.1 Constructivist Grounded Theory

Our research method described in Chapter 5 follows Constructivist Grounded Theory described in Chapter 4. As a reminder, for this study, 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, participant-observation was used for eight projects.

Constantly comparing new codes to existing codes refined the codes and eventually generated categories. Periodic audits checked 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. Memos were written to capture the analysis of codes, examinations of theoretical plausibility, and insights. Constant comparison facilitated the identification of "the conceptual relationship between categories and their properties as they emerged" [20], leading to a resulting descriptive theory. The resulting theory is presented in Section 6.5 and illustrated in Table 6.2. 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 6.1.

6.4.2 Project Context: Project Quattuor

While eight projects were observed in total, this discussion focuses on Project Quattuor. This project shares many similarities with the other projects. To preserve client confidentiality, this study describes Project Quattuor's project as 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 mock-ups 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 the discussion will focus on the iOS team. The first iOS release to the Apple store

Table 6.1: 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."*

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 6.1 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. The plan is consistent with the observed projects with a gradual ramping up of developers to a stable team. Figure 6.2 shows the actual staffing, which is quite different from the plan.

The bar chart on the top of Figure 6.2 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. Five engineers worked for 70% or more of the project's duration. The graph on the bottom of Figure 6.2 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

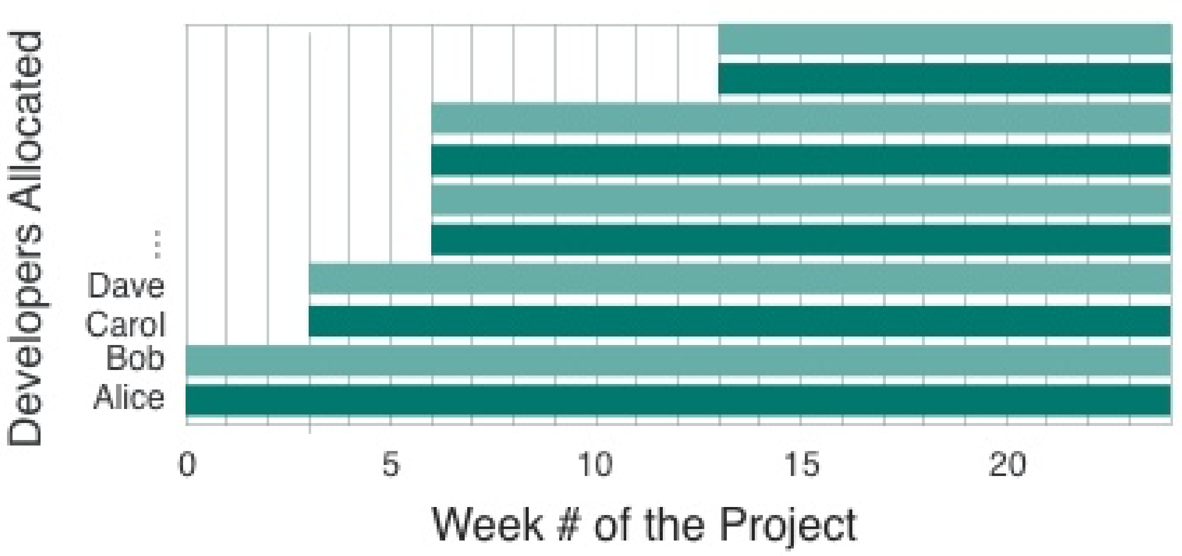


Figure 6.1: Planned Developer Staffing

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 [67]. 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 when the team delivered the first release.

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

6.5 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. The theory of Sustainable Software Development, summarized in Table 6.2, explains how Pivotal teams deliver successful projects despite team disruption. We hypothesize that sustainability emerges from synergistic principles, policies, and practices, which collectively explain how the observed Pivotal

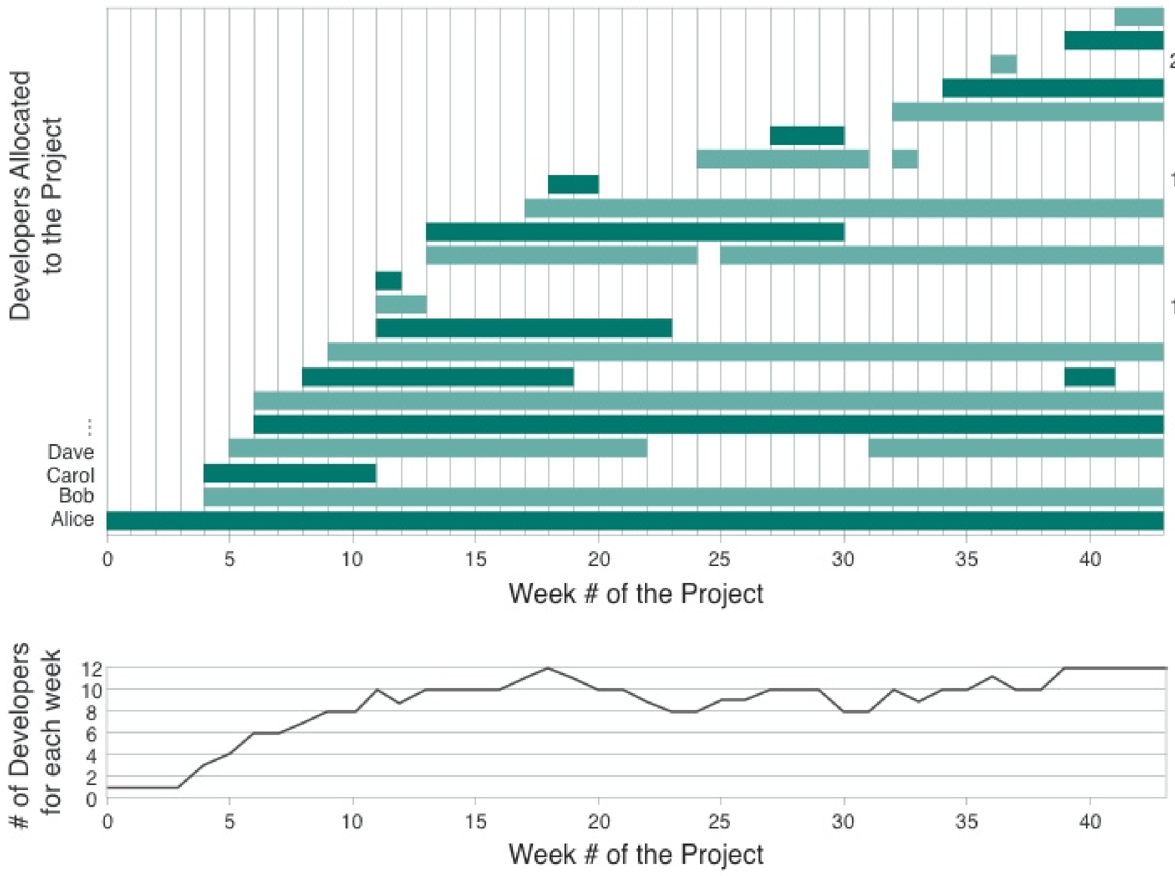


Figure 6.2: Actual Developer Staffing

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.

This section documents each principle, policy, and practice. For each policy and practice, the subsection presents how it is used at Pivotal, and discuss anti-patterns and potential alternatives. Deeper descriptions are provided for practices rarely documented in the literature.

6.5.1 Principles

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 6.2). 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

Table 6.2: Theory of Sustainable Software Development: Principles, Policies, and Practices

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

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.

Encouraging Knowledge Sharing and Continuity

Despite the fresh perspectives added by new team members, team disruption can result 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.

Caring about Code Quality

Enabling knowledge sharing and continuity does not increase the probability of sustainable development if the team starts incurring technical debt [37]. 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.

6.5.2 Policies

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 [56]. 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.

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 losing 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.

Avoid Technical Debt

Description: Technical Debt refers to delaying needed technical work, by taking technical shortcuts, usually in pursuit of calendar-driven software schedules [37].

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

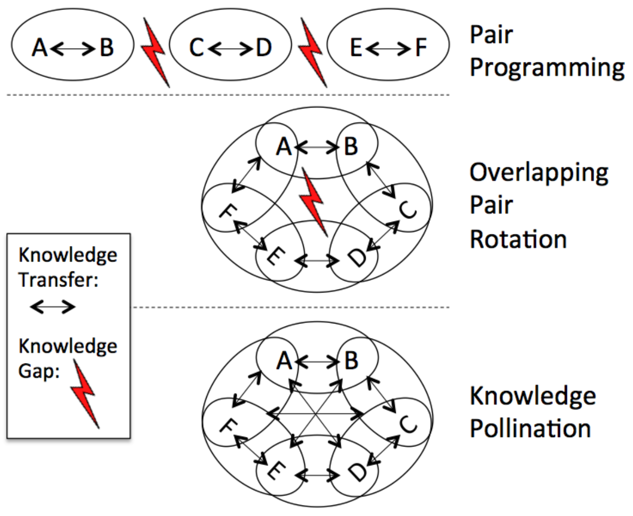


Figure 6.3: Three Levels of Knowledge Sharing

is a dialectic tension [50] 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, one observed team choose to actively pay 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.

6.5.3 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 6.3, where letters A to F represent six developers working in pairs.

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 [75]. When two developers work together, knowledge spreads from one developer to the next [78], as illustrated in Figure 6.3. 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 [9] suggests that this approach would introduce more defects.

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 6.3. 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 understand 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.

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 6.3.

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 the daily stand-ups.

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 [12].

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.

6.5.4 Caretaking the Code Practices

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 write an automated 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 [7]. Later, he refines these ideas as "Test-first programming" [8].

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 [18] or contract testing using mocks [49]. 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.

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."

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" [72].

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." Long running branches is an anti-pattern.

6.6 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?") [63].

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 33 intensive interviews conducted in four different offices, field notes from participant observation on Project Quattuor, and the use of participant-observation on eight projects.

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. Overlapping pair rotation and its supporting principles, policies, and practices are central and unique to the proposed theory. The theory explains why Pivotal teams continue to deliver projects despite team disruption.

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.

6.7 Threats to 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. This risk was mitigated by recording interviews and having another researcher review the results of 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 [21]. This risk was mitigated with another researcher reviewing the coding process.

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 [34]. 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.

6.8 Conclusion

This research 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 33 intensive interviews conducted in four different Pivotal offices, field notes from participant-observation on the Project Quattuor, and the involvement in the eight 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, 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.

Chapter 7

Engendering Team Code Ownership

7.1 Summary

Context: This chapter examines team code ownership, a core category of our theory of sustainable software development presented in the previous chapter. Team code ownership is a software development practice where any team member can modify any part of the team's code. However, many factors beyond official policy affect a developer's sense of ownership.

Objective: The purpose is to understand the factors that affect a team's sense of code ownership.

Method: Following Constructivist Grounded Theory, we conducted a two-year five-month participant-observation of eight software development projects, and interviewed 33 software engineers, interaction designers, and product managers. When team code ownership emerged as a core category in our study, additional data was collected to investigate factors associated with perceived code ownership and related phenomena.

Results: Team code ownership is a feeling. Developers feel team code ownership more when they understand the system context, have contributed to the code in question, perceive code quality as high, believe the product will satisfy the user needs, and perceive high team cohesion.

Limitations: Outcomes of grounded theory research are not statistically generalizable to defined populations, and may not apply to organizations with different software development cultures.

Conclusion: Team code ownership is rooted in numerous cognitive, emotional, contextual and technical factors and cannot be achieved simply by policy.

7.2 Introduction

Team Code Ownership (which is similar to *Collective Code Ownership* and *Shared Code*) is a software development practice where any developer on a team has the right to change any of the team's code. Team code ownership is intended to accelerate development by allowing any developer to fix any team bug and by mitigating delays due to vacations, illness, and other absence [8].

While some research has investigated the effects of different code ownership models, we are unaware of any studies that specifically investigate developers' sense of team code ownership; that is, the complex interactions between developers' knowledge, emotions, and approach to code ownership.

When team code ownership emerged as a core category of our grounded theory study, additional data was collected to investigate factors associated with perceived code ownership and related phenomena.

Participant observation quickly revealed that having the right to change a file does not mean that a specific developer will feel empowered to and justified in making a specific change. For example, a developer may feel reluctant to change code that he or she does not really understand. As the research refined this core finding and drove further data collection, five factors affect feelings of team code ownership.

This chapter consequently reviews existing research connected to team code ownership (Section 7.3), describes some aspects of the grounded theory approach related to code ownership (Section 7.4), and presents the emerging results: five factors associated with team code ownership (Section 7.5). Section 7.6 discusses the study's implications and limitations, followed by a summary of its contributions (Section 7.7).

7.3 Related Work

7.3.1 Team Code Ownership

In Extreme Programming [8], Kent Beck describes a set of interdependent practices for managing feature development and facilitating a collaborative team environment. One of these practices is *collective ownership*—"Anyone can change any piece of code in the system at any time." [7]. The book contrasts collective ownership against "no ownership" and "individual ownership." In 2004, collective ownership is renamed *shared code* [8].

In 2006, Martin Fowler defined *collective code ownership*, similarly to Beck [17], as a contrasting team position to "strong code ownership" where each file has one owner and "weak code ownership" where developers can change files, but an owner keeps an eye on files for which they are responsible.

Later, Bird et al. [9] contrasted the effects of strong- and weak-ownership. They demonstrated that weak ownership leads to more defects than strong ownership for Windows Vista and Windows 7. The study defined ownership for a software component as a percentage of the version control commits for a single developer. They defined a major contributor as someone who has more than 5% of the git commits. A sensitivity analysis revealed that defining strong code ownership within the range from 2% to 10% produced similar results for the study.

Meanwhile, Murphy [41] argued that the concept of code ownership must be unpacked and expanded. He argued that the complexities of code ownership are missed by merely examining git commits to determine who modified which files.

This research defines *team code ownership* as "the ability for any developer on a team to change any of the team's code." For small systems and teams, *collective code ownership* and *team code ownership* are practically synonymous since a member of a four person team usually has the ability to modify any part of the team's code. For a large system with multiple teams, in practice, teams would have strong ownership of their portion of the system. Allowing any pair to modify any part of Microsoft Windows or Pivotal Cloud Foundry is impractical. For a large system with poorly defined team boundaries, it is possible for all the teams to adopt *collective code ownership* without achieving *team code ownership* as one team's changes to the code base may negatively affect other teams.

Team code ownership requires more than a team saying, "everyone can modify anything." Instead, this research examines how a team feels that they own the code. This research defines "sense of team code ownership" as the degree to which individual members of the team feel collective ownership.

7.3.2 Psychological Ownership

Psychological ownership refers to "the feeling of possessiveness and of being psychologically tied to an object" [44]. Targets of ownership, whether physical or immaterial, become the extension of one's self: "What is mine becomes (in my feelings) part of ME" [27]. Ownership can be attached to a part or the whole. Psychological ownership occurs when the object becomes part of the psychological owner's identity. Psychological ownership answers the question, "What do I feel is mine?"

Changes in ownership can have strong effects on our self-identity. An increase in the number of possessions can produce positive effects [16], while a decrease can lead to a personality shrinkage [29]. Someone threatening a person's ownership can trigger strong emotions and responses.

Peirce [44] identifies three sources or "roots" of psychological ownership: "efficacy and effectance, self-identity, and having a place." A major reason for possession of physical goods or abstract ideas is rooted in the innate human desire to be in control; being able to alter one's environment creates feelings of efficacy and pleasure. Ownership fulfills the need for self-identification as people define themselves, express themselves, and ensure their own survival by what they own. Ownership fulfills the need to have a place and a territory to possess. "Each motive facilitates the development of psychological ownership, rather than directly causes this state to occur." Psychological ownership occurs with code because creating software can satisfy the desire for efficacy and effectance, self-identity, and having a place.

Peirce identifies three paths or "routes" to ownership: controlling the target, coming to intimately know the target, and investing the self into the target. With *controlling the target*, targets that can be controlled are perceived to be part of the self. As individuals repeatedly exercise control of an object, eventually this leads to "feelings of ownership toward that object." The higher the autonomy of the job task, the more likely ownership develops toward the activity. When a person has little control over an activity, psychological ownership is unlikely to develop. With *coming to intimately know the target*, the association with the object creates feelings of ownership. One example is when a gardener feels that the garden belongs to the gardener. (This happens routinely with software developers who feel that they own part of the code base, when in reality, the company owns the software.) Feelings of ownership increase as one becomes intimately familiar with the object and associated with it. With "investing the self into the target," people feel that they own what they create, shape or produce. Spending time, energy, and effort enables people to alter their view of themselves to include identity with the object. The more investing in the object, the stronger the psychological ownership. Nonroutine, complex jobs infuse more of individual's ideas resulting in increased ownership.

7.4 Research Method

The research method for this chapter is identical to the research method in Chapter 6. When team code ownership emerged as one of the core categories of the Theory of Sustainable Software Development, additional data was collected in order to identify the factors affecting the sense of code ownership. As an example, Table 7.1 presents some quotes illustrating potential threats that can erode team code ownership for the system context factor. Section 7.5 introduces the factors (system context, code contribution, code quality, product fit, and team cohesion) and are the main contributions for this chapter.

7.5 Team Code Ownership

In the literature, collective code ownership is often treated as a policy statement. In this case, simply claiming that "anyone can modify any piece the code" was not sufficient to engender willingness to

Table 7.1: Examples of Threats for the System Context factor

Threat: Increasing knowledge silos
Sharing knowledge around whole team is important. "Knowledge silos are a red flag for me. For example, we have some siloing around certain parts of the app. Most of it is around the big pain points." "I feel that we don't have that context spread around fully." Hesitancy to jump onto stories without context. "It does make me not completely comfortable to jump into stories on certain aspects of the system. If I was on a story that's going to deal with some tricky part of the system, I would want to be paired with somebody who had more traction on it."
Threat: Increasing team size
With a big team, keeping shared context is challenging as so much parallel work is being done. "Having five, sometimes six pairs on the project means the team is making significant progress each week. It is hard to keep context. If you spend a week on one part of the system, the other pairs are changing the other parts of the system. When you get back to some other place, you don't know what has changed. Because of that speed, it's harder to keep context on everything."
Threat: Increasing code base size
Code continues to expand. "The code is complicated. It keeps on expanding. It is hard to keep it all in my head. It is like the big bang, where it starts out very small and now it just keeps expanding and growing. We are still adding a bunch of features."

modify any file. Rather, ownership is an emotional or qualitative attribute that ties all developers on the team to the project and code base. It is a spectrum where, on one side, each individual has ownership of only their code, and on the other side, everyone on the team owns the entire code base. Some events appear to erode the team's sense of ownership over the project's duration, while some practices appear to counteract these erosions. This section details the five factors that appear most related to team code ownership and examples of events or tendencies that erode it.

7.5.1 System Context

Definition: System context is the knowledge and situational awareness about the code, including the discourse that surrounds the code. System context includes understanding existing design decisions, underlying technologies, the relationship between features and user needs, and the implementation of existing features.

Purpose: Developing an in-depth knowledge of the system exercises the "intimately knowing the target" path of psychological ownership.

For a pair to work efficiently on any part of the system, one of them needs to have enough context to know how that part of the system works. Without enough context, a pair might struggle, slow down, or be blocked in working on a feature.

Team code ownership seems to vary with the context that the developer has about the code; the more the developer knows, the higher the sense of ownership. Knowledge silos, the size of the code base, or the number of developers working in parallel can make it difficult for a programmer to develop a deep system context level.

Threat: Increasing knowledge silos. When developers routinely work on one part of the code base, they can develop specific system context not shared by the team. Code specialization impedes anyone on the team from modifying any part of the team's code. One team said *"we need Marion on that story, only she knows the Apple watch code base,"* and *"Shea knows the ins-and-outs of the legacy integration, we need him to work on this story,"* which means there is a hindering imbalance between the individual and team understanding of the code.

Threat: Increasing team size. We observed the relationship between team size and code context on eight Pivotal projects as a participant-observer. As team size increases, the ability to gain system context decreases. Every day, all pairs are adding to the system. On a five pair team, so much work is happening each day that it becomes increasingly difficult to keep track of everything that changes.

One developer on a ten-person project said, *"I feel that we don't have the context spread around fully. Having five, sometimes six, pairs on the project makes it go really fast, so it's hard to keep context."*

When developers do not have context about part of a system, or context about what remains to be done to finish a story, reluctance to start the next story at the top of the backlog emerges. It's easier to start a story that touches part of the system that they know. As one developer reflected, *"I am not entirely comfortable to jump into stories on certain aspects [of the system]."*

As a coping strategy, one developer, before the start of the work day, skimmed the git commits from the previous day to learn about new classes and changes in design and to understand the features the team added.

As team size grows, there is a potential risk of decreasing an individual developer's sense of team code ownership.

7.5.2 Code Contribution

Definition: Code contribution is the portion of the code that a given developer has worked on.

Purpose: Personally contributing to the code base increases a developer's sense of ownership by exercising "investing in the target" path of psychological ownership.

As a developer works on the code base, the developer's system context level increases. While code contribution level influences the system context level, it is not necessary related: developers might learn about the code through other means different from direct contribution, including conversations at stand-up, impromptu team huddles, or a pair saying *"check out what we did yesterday."*

Threat: Inability to contribute. A developer's inability to contribute to the code base decreases the developer's sense of ownership.

This could happen, for instance, during a pair programming breakdown. When the pairing experience breaks down, one person drives the code development while the partner passively watches. ("Performance Pair Programming" describes when one developer plows through a story and stops listening to the developer's partner.) When one person is writing all the code, individual code ownership replaces team code ownership.

In one situation, the partner took over and ignored the participant's input. The participant reflected, *"I would not be able to explain deeply what we had done. I would not be able to maintain it. I didn't really write it, so I feel very little ownership of it."*

Ideally, Pair Programming is a collaborative experience where both individuals are unable to tell who wrote which portions of the code.

7.5.3 Code Quality

Definition: Code quality relates to how well the code satisfies the project's desirable quality attributes. Desirable quality attributes might include design qualities, performance, reliability, scalability, security, testability, and usability [38].

Purpose: A high quality product satisfies the self-identity motivation of psychological ownership. Developers might not want to be identified with a low quality product.

Low-quality products also tend to involve a disproportionate amount of bug fixes. Developers need a balance between creating new features and fixing bugs each week. Working only on bugs for weeks affects their sense of ownership.

Threat: Pressure to deliver and deprioritizing continuous refactoring. When developers are pressured to deliver more features at the expense of Continuous Refactoring, the code acquires technical debt, the code becomes more difficult to work with, and developers can begin to feel indifferent about the code. When developers begin to experience code apathy, this decreases their sense of team code ownership.

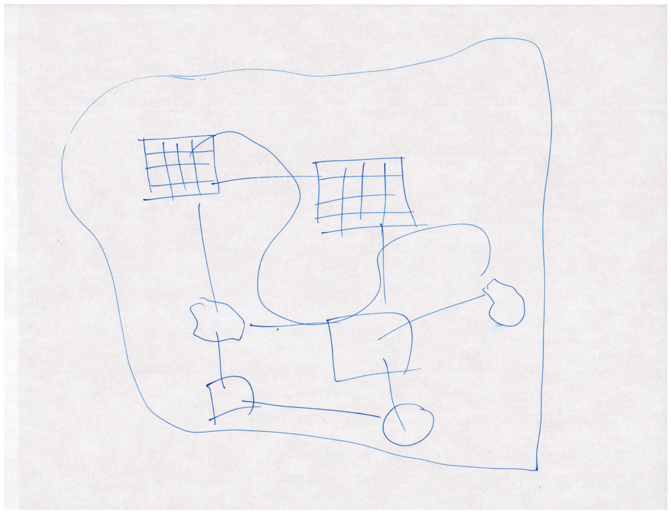


Figure 7.1: "Draw how you feel about the code"

When the team neglects refactoring, new code is simply bolted onto the existing design. Each time the team bolts something else on, bolting on the next piece becomes more complicated. Thus, a dilemma arises for the programmers working on the next story that touches this part of the code: do they continue bolting on more code, or do they perform the pretermitted refactoring? A team's avoidance of refactoring may be a sign that code apathy is settling in. Code apathy results in reduced quality, as the developers become less invested in the craftsmanship of the code.

One developer felt *"proud and disgusted"* about the code base. He is simultaneously proud of each refactoring that the team performed and disgusted by the technical debt the team accrued by taking shortcuts to ship more features. The developer drew Figure 7.1 to show his feeling about the code, *"it is generally orderly with a few bits that maybe are not as orderly."*

Before the first launch of a product, the product manager suggested that the team deliver more features at the expense of technical debt. For some of the team, this was an unacceptable tradeoff, and those developers decided not to cut corners. Others on the team complied with the request and incurred technical debt. The entire team ended up paying the consequences with extensive refactors after the launch. On a communal code base, one pair adding tech debt affects everyone on the team.

When code apathy settles in, team members adopt the attitude that someone else will solve the problem with the code. When this attitude permeates a team, no one is solving the problems.

The team wants to feel pride in improving code quality. It feels good to be improving the code design and readability. If the team starts neglecting these concerns, it can engender a sense of disgust and apathy for the code can spread throughout the team.

7.5.4 Product Fit

Definition: Product fit is developers believing that features of the product will satisfy the user's needs.

Purpose: The observed engineers want to create products that matter to the users. Delivering a product that matters to someone satisfies the self-identity motivation of psychological ownership.

Threat: Ignoring user feedback. When the product manager ignores feedback from user research and usability testing, developers may lose faith in the product's ability to achieve its goals. Developer motivation and engagement can decrease when developers perceive they are building a feature that users have explicitly said they do not want yet is built to solve a business goal.

Threat: Ignoring developers feedback about the product. Pivotal's balanced team approach is founded on collaboration between product managers, interaction designers, and developers. When product managers or other stakeholders ignore feedback from developers, developers can begin to feel less ownership in the product, and in turn, be less motivated to work on the project.

Feature apathy or product apathy can result in a poorly crafted product that does not meet the customer's needs.

7.5.5 Team Cohesion

Definition: Team cohesion is the degree to which team members identify as part of the team, stick together through adversity and take pride in the team's accomplishments [10, 6, 74].

Purpose: Team cohesion satisfies the "having a place" motivation of psychological ownership.

Threat: Distancing a developer from the team. Team apathy manifests when developers do not feel that they are a part of the team. Developers feel less ownership of the code base when they feel excluded from the team.

Several behaviors were observed that can distance a developer from the team: interrupting the developer during discussions, using poor listening skills so that the developer feels unheard, or talking beyond the developer's level of technical expertise.

On one team, during discussions, the team talked about code but never looked at the source code. One developer found these abstract discussions difficult to follow. Sometimes the team discussed parts of the code that the individual had not seen recently. When the team discussed two variants of coding practices without showing concrete examples, the programmer could not contribute. When the developer raised this issue to the team and the team continued with the status quo, the programmer felt marginalized by the team.

Poor onboarding of developers can contribute to feelings of isolation. On one project, there was a time crunch and the team was feeling the pressure to deliver stories. When the team added developers, the team had a "sink or swim" attitude, letting new team members figure things out on their own, hence making them feel unwelcome.

When developers feel that the team does not care about them, their sense of ownership can decrease.

7.6 Discussion

7.6.1 Transitioning to team code ownership

The above results have numerous implications for teams attempting to transition to team code ownership. Some developers effortlessly make the transition to team code ownership. They immediately see the benefits of being able to modify any part of the code base and quickly shift from "I made this" (personal ownership) to "we made this" (collective ownership.)

Others may struggle with team code ownership for several reasons:

- Developers may struggle to transition to a caretaker mindset. In one interview, a software engineer struggled to describe the developer's relationship with the code on a very challenging project and settled in on the caretaker metaphor: *"Sometimes I kind of feel like a janitor to [the code base]. Maybe caretaker would be better. Yeah, probably caretaker. I feel like a janitor just cleans up messes, but a caretaker makes things better."*
- A developer may be distraught at *"seeing my work slowly removed from the app."*
- Developers can no longer take pride in functionality that they exclusively develop.
- Existing knowledge silos, which hinder team code ownership, may be slow to break down.

New hires struggling with the transition slowly realize that *"someone else is going to take over and they're going to do fine. I can move onto something else and that's okay."* They recognize the lack of long-term individual authorship, learn to expect their code to be transitory, develop trust in their teammates and thus loosely hold personal contributions. *"The code that I write today may be in the code base for a little while, and it will evolve into something better."* Eventually, they experience the benefits of a collaborative environment: *"People are a lot more flexible all across the board, with changing things or accepting feedback or collaborating,"* and the team can say *"Hey, this is our code!"*

Shifting from individual to team code ownership may require multiple and complementary practices to actively remove knowledge silos. In this case, daily pair rotation helped combat knowledge silos. Moreover, for developers with strong individual ownership tendencies, sharing ownership first with a small group (where trust and communication come easier) may help. One Pivotal engineer uses improvisation and collaboration games to help teams practice letting go of control, trusting the team, and learning to be pleasantly surprised by what emerges.

7.6.2 Results Evaluation

The factors influencing team code ownership presented in Section 7.5, have emerged from the Grounded Theory research study introduced in Section 6.5. While other factors may influence team code ownership, this discussion focuses only on those that were observed during the study. Grounded Theory studies can be evaluated using the following criteria [11]:

Credibility: The 33 intensive open-ended interviews and numerous field notes from participant-observation serve as a rich and credible data set for the analysis.

Originality: The research broadens the idea of team code ownership by acknowledging that collective code ownership is more than a policy statement, and by uniquely identifying factors that affect the team's sense of code ownership.

Resonance: Several participants reviewed the findings and indicated that both the factors and threats resonate with their experience.

Usefulness: The study identifies factors associated with ownership and suggests several ways of engendering team code ownership.

This work analyzed software projects at the Silicon Valley office of Pivotal following Extreme Programming. From an **external validity** perspective, grounded theory is non-statistical, non-sampling research. The results therefore cannot be statistically generalized to a population. Rather, researchers and professionals can adapt the concepts and ideas to other contexts case-by-case.

Finally, the results might be influenced by **researcher bias** or **prior knowledge 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. While a participant-observer gains perspective an outsider cannot, an outside observer might see something a participant observer will miss. Similarly, while prior knowledge helps the researcher interpret events and select lines of inquiry, prior knowledge may also blind the researcher to alternative explanations [21]. These risks were mitigated by recording interviews and having the other researchers review the coding process.

7.7 Conclusion

This chapter reports results from a participant-observation, constructivist grounded theory study at Pivotal, a large American software company employing Extreme Programming practices. It provides three main contributions.

1) The observations clearly indicate that **team code ownership is a feeling to be engendered not a policy to be decreed**.

2) Meanwhile, both discussions with and observations of participants suggest five factors associated with strong feelings team code ownership. Pivotal developers more acutely feel team code ownership when i) they understand the system context; ii) they have contributed to the code in question; iii) they perceive code quality as high; iv) they believe the product will satisfy user needs; and v) they perceive team cohesion as high.

3) Moreover, diverse events and trends can undermine sense of ownership, including: increasing knowledge silos, increasing code base size, increasing team size, inability to contribute, pressure to deliver and deprioritizing continuous refactoring, ignoring user feedback, ignoring developer feedback, and distancing a developer from the team.

In conclusion, Pivotal's developers find team code ownership highly advantageous; however, transitioning to a team code ownership model is easier for some than others. Some agile practices including continuous pair programming, overlapping pair rotation, continuous refactoring, and test driven development appear to help. Promising angles for future research include more nuanced explorations of the code ownership spectrum, further exploration of the roles of emotion and identity, as well as developing specific practices for facilitating ownership transitions.

Chapter 8

Removing Waste

8.1 Summary

Context: This chapter examines software waste, as a core category of our theory of sustainable software development presented in Chapter 6. Since software development is a complex socio-technical activity that involves coordinating different disciplines and skill sets, it provides ample opportunities for waste to emerge. Waste is any activity that produces no value for the customer or user.

Objective: The purpose is to identify and describe different types of waste in software development.

Method: Following Constructivist Grounded Theory, we conducted a two-year five-month participant-observation study of eight software development projects at Pivotal, a software development consultancy. We also interviewed 33 software engineers, interaction designers, and product managers, and analyzed one year of retrospection topics. We iterated between analysis and theoretical sampling until achieving theoretical saturation.

Results: This chapter presents the first empirical waste taxonomy for software. It identifies nine wastes and explores their causes, underlying tensions, and overall relationship to the waste taxonomy found in Lean Software Development.

Limitations: Grounded Theory does not support statistical generalization. While the proposed taxonomy appears widely applicable, organizations with different software development cultures may experience different waste types.

Conclusion: Software development projects manifest nine types of waste: building the wrong feature or product, mismanaging the backlog, rework, unnecessarily complex solutions, extraneous cognitive load, psychological distress, waiting/multitasking, knowledge loss, and ineffective communication.

8.2 Introduction

"The engineers are depressed. The project grinds them down... It is hard to know which problem to tackle first. There is coupling everywhere... Each layer of the system has unnecessary complexity... The depth of knowledge about the system is meager... There is a lot of waiting... Building the Java code takes ten minutes. Starting the server takes seven minutes. Running the Javascript tests take two minutes. Running the integration tests take 47 minutes. Continuous integration takes forever to run all the tests and get the code onto the acceptance environment.

There is waste everywhere." —Software Engineer on Project Septem.

Software development is a complex socio-technical activity that involves coordinating different disciplines and skill sets. Identifying user needs, crafting features for those needs, identifying and prioritizing value, implementing features, releasing, and supporting products provide ample opportunity for waste to creep in.

Here, "waste" refers to "any activity that consumes resources but creates no value" for customers [76]. Reducing waste, by definition, improves efficiency and productivity. Waste is like friction in the development process.

However, reducing waste is difficult not least because *identifying* waste is difficult. Numerous cognitive phenomena, including status quo bias [31], hinder practitioners' propensity and ability to notice waste in existing practices. Identifying the types of waste that often occur in software projects may, therefore, facilitate reducing waste. Identifying and eliminating waste is a key principle of lean manufacturing.

The Toyota Production System [42, 60] transformed manufacturing from batch-and-queue to just-in-time. The similarities between batch-and-queue and waterfall software development, as well as just-in-time and iterative software development, inspired several software development methods [45, 3]. These methods adapt, in a top-down fashion, lean principles for software environments.

However, manufacturing differs from software development in significant ways. Manufacturing produces physical products; software is intangible. While the 1000th car costs about as much to make as the 999th car, the marginal cost of the 1000th copy of a mobile app is near zero. While most factories build batches of near-identical goods, much software remains unique. Typically, manufactured products evolve much slower than software.

Given the obvious differences between developing software and manufacturing physical products, software development may entail waste types never envisioned in lean manufacturing. Even the most careful adaptation of lean principles for software may not have identified such waste types. Therefore, we conducted an in-depth, longitudinal investigation of a successful software company to address the following research question:

Research Question: "What types of waste are observable in software development?"

Next, Section 8.3 summarizes the history of lean and review related work. Section 8.4 describes the research method. Section 8.5 presents the emergent waste taxonomy. Section 8.6 compares this model with the waste list from Lean Software Development. Sections 8.7 and 8.8 evaluate the results, describe limitations, and conclude the chapter.

8.3 A Brief History of Lean

Lean Thinking is a concept proposed by Womack [76] following his analysis of The Toyota Production System. The Toyota Production System prioritizes waste removal by creating a culture that pursues waste identification and elimination in the entire production of a vehicle [42, 60]. In 1945, Toyota optimized for the production rate of each system, keeping like machines near each other. Ohno rearranged equipment so that the output of one machine fed into the next machine, slowed machines down to have the same cadence, and only produced material when it was needed. After optimizing Toyota's factories, Toyota then trained their suppliers so that the entire production of a vehicle was just-in-time, transforming from mass production to lean production. The resulting "pull" system was easy to reconfigure, minimized inventory, and supported short production runs.

Lean Thinking describes a process of identifying and removing waste in a value stream [76]. The process discerns three types of activities: activities that clearly create value; activities that create no value for the customer but are currently necessary to manufacture the product; and activities that create no value for the customer, are unnecessary, and therefore should be removed immediately; i.e., waste.

The Toyota Production System characterized seven types of manufacturing waste [60] shown in Table 8.1. Later, Womack and Liker each added a waste type: value and non-utilized talent [76, 36].

Mary and Tom Poppendieck created Lean Software Development [45] by adapting Lean Thinking and the Toyota Production System from manufacturing to software development. Table 8.2 presents their comparison of manufacturing waste with software waste.

Adapting a taxonomy from a reference discipline (e.g. manufacturing) for a target discipline (e.g. software engineering) manifests at least four threats to validity:

1. The target domain may include concepts (wastes) not found in the source domain.
2. The source domain may include concepts not found in the target domain.
3. Concepts from the source domain may bias our perception of superficially similar but fundamentally different concepts in the target domain.

Table 8.1: Toyota Production System Definition of Manufacturing Waste

Waste Type	Description
Inventory	The cost of storing materials until they are needed. The material might never be used.
Extra Processing	The cost of processing that is unneeded by a downstream step in the manufacturing process. (Sometimes an inefficiency from not seeing the entire process.)
Overproduction	The cost of producing more quantity of components than necessary for the present.
Transportation (of goods)	The cost of unnecessarily moving materials from one place to another place.
Waiting	The cost of waiting for a previous upstream step to finish.
Motion (of people)	The cost of unnecessary picking up and putting things down.
Defects	The cost of rework from quality defects.
Value (added by [76])	The cost of producing goods and services that do not meet the needs of the customer.
Non-utilized Talent (added by [36])	The cost of unused employee creativity and talent.

Table 8.2: Comparison of Manufacturing Waste with Lean Software Development Waste

Toyota Production System's Manufacturing Wastes	Lean Software Development Wastes [46]
Inventory	Partially Done Work
Extra Processing	Relearning
Overproduction	Extra Features
Transportation (of goods)	Handoffs
Waiting	Delays
Motion (of people)	Task Switching
Defects	Defects
Value (added by [76])	N/A
Non-utilized Talent (added by [36])	N/A

4. The organization of concepts in the source domain may not fit the target domain (e.g., two or more manufacturing wastes might map into a single software engineering waste or vice versa).

It is, therefore, incumbent upon researchers to empirically evaluate concepts, taxonomies, and theories adapted from reference disciplines. We are not aware of any direct empirical validation of the Lean Software Development waste taxonomy; this motivates the current study.

That said, several studies have used the Lean Software Development waste model. For example, Power and Conboy combine it with literature in manufacturing, lean production, product development, construction, and healthcare. They shift from using wastes of inefficiencies to impediments to flow [47].

Several studies applied Value Stream Mapping to software development. Value Stream Mapping, popularized by Womack, systematically examines each stage for waste. Interestingly, these studies only found *waiting* waste generated in a batch-and-queue system [2, 33, 40]. One study identified the wastes of motion and extra processing from interviews, not the current state map [40]. These studies typically reduced waste by switching the organization from waterfall to iterative software development or reducing the batch size in iterative software development [2, 33, 40].

8.4 Research Method

Chapter 5 described the research method. Once *removing waste* emerged as a core category, we collected and analyzed data from retrospection meetings. We continued theoretical sampling for removing waste in additional interviews and participant observations until no further waste-related categories were evident, i.e. theoretical saturation.

- Waste Category: Psychological Distress
  - Cause Property: Low team morale
    - Retro Topic: Frustrated developers
    - Retro Topic: Not managing expectations
    - Retro Topic: Negative attitudes
    - Retro Topic: Apathy
    - Retro Topic: Not knowing everyone on the team
    - Retro Topic: Project feels like it is falling apart emotionally
    - Retro Topic: Unacknowledged by management
    - Retro Topic: Messy code decreasing sense of ownership
  - Cause Property: Rush mode
    - Retro Topic: Fixed set of features with a fixed timeline
    - Retro Topic: Aggressive timelines
    - Retro Topic: Shifting deadline
    - Retro Topic: Scope creep
    - Retro Topic: Repeatedly hearing “This is due today”
    - Retro Topic: Long days
    - Retro Topic: Overtime
  - Cause Property: Interpersonal or team conflict
    - Retro Topic: Criticizing in public
    - Retro Topic: Difficult pairings
    - Retro Topic: Pairing fatigue
    - Retro Topic: Not listening
    - Retro Topic: Interpersonal conflict

Figure 8.1: Waste Organization Example (Psychological Distress)

8.5 Results: Types of Waste in Software Engineering

We identified nine types of waste (Table 8.3). This section defines, elaborates, and provides examples of each type, including associated tensions where available.

8.5.1 Waste: Building the Wrong Feature or Product

Building features (or worse, whole products) that no one needs, wants, or uses obviously wastes the time and efforts of everyone involved. We observed this waste affecting team morale, team code ownership [56], and customer satisfaction.

The product features for Project Ses were designed based on a given persona—i.e. a fictional, archetypal user [26]. However, consulting several real intended users revealed that the persona was deeply flawed as the users did not need the product. The intended users invalidated the persona. Building the intended product is risky and probably wasteful.

Tension: User needs versus **business wants.** Some projects exhibit a tension between user needs and business goals. Practitioners may struggle to produce something that simultaneously satisfies the users and the business.

On Project Quattuor, the client wanted to add a news feed to a mobile phone application that controlled a real world product in order to increase marketing awareness. However, user validation revealed that no users wanted this feature, and several reacted quite negatively. Despite numerous conversations, the marketing department insisted on adding the feature.

8.5.2 Waste: Mismanaging the Backlog

The product backlog can be mismanaged in several ways, leading to delays of key features or lower team productivity.

On several projects, we observed engineers working on low-priority stories through "backlog inversion." This occurs when the engineers working through the backlog get ahead of the product manager who is prioritizing the backlog. For instance, the product manager might prioritize the next ten stories in the backlog, but the engineers get to story 15 before the product manager gets back to prioritizing. This creates waste as engineers implement potentially outdated, low-value, or even counterproductive stories ahead of high-value stories.

Mismanaging the backlog can also lead to duplicated work. We observed duplicate stories in the backlog, two engineers working on the same story because one had forgotten to change its status, and two

Table 8.3: Types of Software Development Waste

Waste Description	Observed Causes
Building the wrong feature or product The cost of building a feature or product that does not address user or business needs.	User desiderata (not doing user research, validation, or testing; ignoring user feedback; working on low user value features) Business desiderata (not involving a business stakeholder; slow stakeholder feedback; unclear product priorities)
Mismanaging the backlog The cost of duplicating work, expediting lower value user features, or delaying necessary bug fixes.	Backlog inversion Working on too many features simultaneously Duplicated work Not enough ready stories Imbalance of feature work and bug fixing Delaying testing or critical bug fixing Capricious thrashing
Rework The cost of altering delivered work that should have been done correctly but was not.	Technical debt Rejected stories (e.g. product manager rejects story implementation) No clear definition of done (ambiguous stories; second guessing design mocks) Defects (poor testing strategy; no root-cause analysis on bugs)
Unnecessarily complex solutions The cost of creating a more complicated solution than necessary, a missed opportunity to simplify features, user interface, or code.	Unnecessary feature complexity from the user's perspective Unnecessary technical complexity (duplicating code, lack of interaction design reuse, overly complex technical design created up-front)
Extraneous cognitive load The costs of unneeded expenditure of mental energy.	Suffering from technical debt Complex or large stories Inefficient tools and problematic APIs, libraries, and frameworks Unnecessary context switching Inefficient development flow Poorly organized code
Psychological distress The costs of burdening the team with unhelpful stress.	Low team morale Rush mode Interpersonal or team conflict
Waiting/multitasking The cost of idle time, often hidden by multi-tasking.	Slow tests or unreliable tests Unreliable acceptance environment Missing information, people, or equipment Context switching from delayed feedback
Knowledge loss The cost of re-acquiring information that the team once knew.	Team churn Knowledge silos
Ineffective communication The cost of incomplete, incorrect, misleading, inefficient, or absent communication.	Team size is too large Asynchronous communication (distributed teams; distributed stakeholders; dependency on another team; opaque processes outside team) Imbalance (dominating the conversation; not listening) Inefficient meetings (lack of focus; skipping retros; not discussing blockers each day; meetings running over (e.g. long stand-ups))

engineers independently addressing the same pain point (e.g. making the build faster) by not communicating what they were doing in the backlog.

Tension: Writing enough stories versus **writing stories that will never be implemented.** Pivotal product managers attempt to provide the team with a steady stream of ready, high-value work. This creates a tension between writing enough stories for the team to work on and "over-producing" stories that might never be implemented. Writing too few stories causes the team to idle while writing too many stories wastes the product manager's time. We observed teams running out of work on rare occasions; we did not observe product managers writing too many stories.

Tension: Finishing features versus **working on too many features simultaneously.** Product managers decompose a feature into a set of stories and typically aim to create the minimal viable product as quickly as possible by sequencing the stories to finish just enough of each feature before starting another feature.

On Project Quattuor's backend system, we observed one product manager starting too many tracks of work at once by prioritizing a breadth of features instead of finishing started features. Unfortunately, several tracks of work were not completed by the first release date. The work in progress was disabled with feature flags. Starting work, changing priorities, and halting work in flight can result in waste.

We observed that teams usually prefer to maintain a shippable product while rapidly finishing the simplest possible version of each new feature.

Tension: intransigence versus **capricious adjustments.** Responding to change quickly is a core tenet of agile development and often thought of as the opposite of refusing to change. However, responding to change is more like a middle ground between intransigence (unreasonably refusing to change) and thrashing (changing features too often, especially arbitrarily alternating between equally good alternatives).

On Project Kvin, for example, the launch was delayed while the business fiddled with the sequence and number of steps in the user registration process. Project Duo was similarly delayed by a product manager repeatedly resequencing an order customization process.

8.5.3 Waste: Rework

From a Waterfall perspective, one might classify any revision of existing code as "rework." This problematically fails to distinguish between situations where things could have been done right based on the information available then from situations where new information reveals a better approach.

Contrastingly, our participants classify revising work that should have been done correctly but was not as *rework*, and improving existing work based on new information as *new work*. *Rework* wastes time and resources by definition. We observed numerous sources of *rework* including technical debt, defects in work products, poor testing strategy, rejected stories, stories with no clear definition of done, and ambiguous mock-ups.

Technical debt refers to the risks of delaying needed technical work, by taking technical shortcuts, usually to meet a deadline [37]. These shortcuts are waste and often burdens the team later, as described in the *extraneous cognitive load* waste. On Project Quattuor, engineers felt pressured to deliver stories quickly and skipped refactoring, resulting in many weeks of *rework* after the first release.

Defects and bugs in the code, stories, mock-ups, and code result in *rework*. On every project, we observed defects. On several occasions, mistakes in stories and acceptance criteria resulted in engineering *rework*. On Project Quattuor, the interaction designers created mockups optimized for English, not the target language. After implementing the application, the team realized that the target language text needed more space than the English translations, requiring *rework* for several design components. On Project Kvin, the interaction designer forgot to consider mobile phones when creating the mock-ups. After building a few screens, the team realized that the website did not work well on mobile devices requiring *rework*.

Rejected stories—stories that a product manager rejects delivered work because the implementation does not satisfy the acceptance criteria—requires *rework* as the developers need to fix the delivered work.

Stories with no clear definition of done (e.g. stories with ambiguous acceptance criteria or ambiguous mock-ups) resulted in *rework*. On Project Ses, the engineers showed a finished story to the interaction designer for feedback. The interaction designer pointed out a missing interaction, which was neither in the story nor the mock-up

8.5.4 Waste: Unnecessarily Complex Solutions

Unnecessarily complex solutions can be caused by feature complexity, technical complexity, or lack of reuse. Unnecessary feature complexity wastes users' time as they struggle to understand how to use the system and achieve their objectives, e.g. requiring the user to fill in form fields not related to the task at hand. Some features bring unnecessary technical complexity since a simpler interaction design would have solved the same problem.

On Projects Tes, Ses, and Septem, complicated legacy components were refactored into simpler, easier to understand components. However, personal and organizational goals may misalign on this issue—one Pivotal engineer complained that a client engineer's attitude was, *"the more complicated, the better, as that means my role is more important"* —Participant 29.

Another way to increase system complexity is through a lack of reuse, i.e., building a new component instead of reusing an existing one. In code, lack of reuse can manifest as duplicated code and similar components that have similar functionality. In mockups, lack of reuse produces "snowflake designs," interaction designs which do not take advantage of design reuse, e.g. two unique user interaction flows that could be unified into similar experiences or two visual components that solve the same concern.

On Project Duo, the interaction designer created a left-to-right navigational flow for configuring the product but designed a top-to-bottom navigational flow for the checkout page. Both sequences allowed the user to change a previous choice, jump to the correct page, and invalidate dependent information. In retrospect, using the same interaction design treatment for both would have been faster.

On Project Quattuor, multiple interaction designers produced different design treatments for the same concept. The product shipped multiple versions of layouts, lists, alerts, and buttons. In some cases the team implemented expensive interactions to delight users but the interactions were not reused.

On Project Kvin, the interaction designer created two sets of form inputs which necessitated multiple CSS styles for the HTML form input tags. Singular designs require engineering to build unique solutions with no possibility of reuse.

Tension: Big design up-front versus **incremental design.** Many projects exhibit a tension between up-front and incremental design. Rushing into implementation can produce ineffective emergent designs, leading to rework. However, big up-front design can produce incorrect or out-of-date assumptions and inability to cope with rapidly changing circumstances, also leading to expensive rework. The desire to avoid rework and differing development ideologies, therefore, motivates the tension and disagreement over big design up-front versus incremental design.

The observed teams expected the product features to change even when the client had clearly defined the project. On all projects with interaction designers, after the interaction designer conducted user research and discovered new information about the user's needs, the feature set changed. No amount of up-front consideration appears sufficient to predict user feedback. The observed teams preferred to incrementally deliver functionality and delay integrating with technologies until a feature required it. For example, an engineer would only add asynchronous background jobs technology when working on the first story that requires the needed technology, even if the team knew it would need the technology at the project's beginning.

We observed teams using common architectural and design solutions from similar, previous projects without explicit architectural or design phases.

8.5.5 Waste: Extraneous Cognitive Load

Cognitive Load Theory [65] posits that our working memory is quite limited and overloading it inhibits learning and problem solving [4]. Intrinsic cognitive load refers to the innate complexity of the task, while extraneous cognitive load refers to the cognitive load unnecessarily added by the task environment, or the way the task is presented [66]. Reducing the burden on working memory by removing extraneous cognitive load is therefore associated with more efficient learning among other positive outcomes [69].

Since many software development activities have high intrinsic cognitive load and developer's mental capacity is a limited resource, we view extraneous cognitive load as waste. While we cannot observe cognitive load directly, we did observe sources of extraneous cognitive load including overcomplicated stories, ineffective tooling, technical debt, and multitasking.

Overcomplicated stories—user stories that are unnecessarily long, complex, unclear, or replete with pointers to other necessary information—are precisely the sort of task materials that Cognitive Load Theory predicts will overburden working memory. On Project Quattuor, one story modifying the presentation of status resulted in the pair creating a spreadsheet listing out the complex behavior. The logic had become too complex to reason about in an individual's working memory. The team, concerned about code maintenance and readability, asked the product manager if simplifying the logic was possible.

Ineffective tooling includes convoluted, nonfunctional, premature, complicated, unstable, outdated, unsupported, time-consuming, or inappropriate-for-the-task software libraries, as well as poorly designed development environments and deployment processes. Participant 13 said that one arcane technology *"makes me angry enough that I want to hack into it, expose how useless and horrible it is, and wipe this miserable product off the face of the earth!"*

Technical debt introduces the risks of the code being harder to understand and modify. We observed teams suffering from technical debt with long-running, existing code bases. On Project Tes, for example, running the test suite produced 87,000 lines of output, including deprecation warnings, exceptions, and test noise. Engineers ignored the overwhelming output which contained important information. On Project Ses, dead code littered the code base along with convoluted objects. Project Septem suffered from engineers introducing an idea in one part of the code base, but not applying the concept systematically. These examples illustrate how technical debt creates more things developers need to remember: in other words, how technical debt increases extraneous cognitive load.

Multitasking—performing two or more activities simultaneously or rapidly alternating between them—increases cognitive load as the multitasker attempts to hold two or more sets of information in working memory or needs to unnecessarily re-load the information into working memory. While observed engineers prefer to finish one task before beginning the next task, they also try to convert excessive waiting (e.g. long builds, long tests, waiting for feedback) into productive time by multitasking (see *waiting/multitasking* waste description for more detail).

8.5.6 Waste: Psychological Distress

"Stress is the nonspecific response of the body to any demand made upon it" [59]. Stress may be beneficial (eustress) or harmful (distress). We see psychological distress as a kind of waste for the same reason as *extraneous cognitive load*: developers are a limited resource, which distress consumes. Job-related psychological distress causes absenteeism, burnout, lower productivity, and a variety of health problems [73].

On Project Quattuor, for example, the team rushed to release a fixed feature set by a fixed date. Daily, the team decreased a countdown written on an office whiteboard to the release date. We observed low team morale, rush mode, lack of empathy, and waiting too long to resolve interpersonal issues leading to people working inefficiently. Furthermore, the team felt that over-emphasizing the deadline was increasing stress and leading to poor technical decisions and eventually erased the countdown from the whiteboard. Participants felt that fixing both scope and schedule was antithetical to Pivotal's software process, where the client either chooses the release date and gets only the features ready by then or chooses key features and ships the product when the features are ready.

8.5.7 Waste: Waiting/multitasking

Having developers waiting around, working slowly, or working on low-priority features because something is preventing them from proceeding on high-priority features wastes their time and delays their projects. For example, we observed developers waiting on (or looking for) product managers and designers to clarify a story's acceptance criteria. On Project Quattuor, product managers started multitasking while accepting stories because the acceptance environment was unreliable. We also saw team members waiting around because of missing video-conferencing equipment.

Ohno described *waiting* waste as hidden waste since people start working on the next job instead of waiting [42]. The Toyota Production system exposes *waiting* waste by requiring someone to pull the red cable to halt the production line. On Project Ses, it took 58 minutes to run the build locally and 17 minutes on the build machine due to parallelization on four machines. Team members would push code as a branch to the build machine instead of running tests locally. While the build machine ran the tests, the engineers would either wait or context switch onto different work. If the branch passed, some time later, they would merge their code into the team's code. If the branch failed, the engineers would decide either to finish the work that they were doing or to switch back and fix the issue. Some engineers found the context switching exhausting. This "solution" to avoid *waiting* created *extraneous cognitive load* waste.

Tension: Wait, block, or guess. When needed information is missing, engineers appear to have three options: 1) wait for the information, 2) suspend (block) the story and work on something else, or 3) act without the information. The best option depends on how far into the story the pair is, how long they have to wait, and their confidence in their guess.

Tension: Waiting versus **context switching.** When possible, engineers would often use waiting time to attempt to remedy problems or reduce the duration of future waiting (e.g. shorten the build). When this was not possible, engineers often worked on something else instead of idling. Unfortunately, task switching decreases productivity and increases mistakes [39]. For short waits, taking a break (e.g. playing table tennis) may be less wasteful than switching to another task.

8.5.8 Waste: Knowledge Loss

Knowledge loss occurs when a team member with unique knowledge leaves a team or company—the latter being a more extreme form of *knowledge loss.*

In projects with knowledge silos, team churn leads to wasted effort as the team regains lost knowledge. For legacy systems, we observed teams sleuthing the code base, commit messages, and completed stories in the backlog to understand the code.

On Project Octo, a complete team turnover required the new team to spend months understanding the system, during which the team's velocity was practically zero.

The observed teams reduced knowledge loss by actively removing knowledge silos and caretaking the code by adopting the principles, policies, and practices of Sustainable Software Development [57]. Teams promoting knowledge sharing appear less susceptible to knowledge loss from team churn or team member rotation.

8.5.9 Waste: Ineffective Communication

Ineffective communication is incomplete, incorrect, misleading, inefficient, or absent communication. We observed that large team sizes, asynchronous communication, imbalance in communication, and inefficient meetings reduced team productivity.

We observed issues with asynchronous communication on Project Quattuor. The team was distributed between two offices separated by an hour commute. We observed the team using remote pairing and

Table 8.4: Comparison to Lean Software Development Waste

Software Development Wastes	Lean Software Development Wastes
Building the wrong feature or product	Extra features
Mismanaging the backlog	Partially done work
Rework	Defects
Unnecessarily complex solutions	Not described
Extraneous cognitive load	Not described
Psychological distress	Not described
Waiting/multitasking	Delays Task switching
Knowledge loss	Relearning
Ineffective communication	Not described
Not observed	Handoffs

engineers commuting between the offices to mitigate the effects of a large distributed team, but communication issues continually arose in the retrospections.

On Project Ses, we observed that one person dominated meetings, which prevented quieter personalities from sharing their perspectives.

On Project Quattuor, when the project started, the iOS team was not effectively reflecting on its process to make informed decisions. Adding weekly retros helped the team reflect and respond to problems. Over several weeks, the remaining teams added their own retros.

8.6 Comparing to Lean Software Development

This section compares and contrasts our waste taxonomy (Section 8.5), with Lean Software Development's waste taxonomy [46] category by category.

While several categories are analogous (Table 8.4), we observed four types of waste not found in Lean: *unnecessarily complex solutions*, *extraneous cognitive load*, *psychological distress*, and *ineffective communication* (see Section 8.5 for details). Meanwhile, we did not observe Lean Software Development's *handoff* waste type.

Handoffs: We did not observe *handoff* waste—the loss of tacit knowledge when work is handed off to colleagues—as Pivotal follows an iterative software development process with cross-functional teams. We did observe *waiting* waste as engineers might contact people outside the team who had needed information. *Handoffs* certainly contribute to the wastes of *knowledge loss*, *ineffective communication*, and *waiting*. However, our data does not support *handoffs* as a *type* of waste.

Building the wrong feature or product and **Extra features**: In our model, *building the wrong feature or product* describes not addressing user or business needs. Pivotal's process relies on user validation to assess value in solving the user's needs and iterating from a minimal viable product. In Lean Software Development, the *extra features* waste describes adding in features that are not necessary for the user's current needs. Both perspectives align on delaying features until necessary. *Extra features* does not cover the waste from missing important business needs. *Building the wrong feature or product* subsumes *extra features*.

Mismanaging the backlog and **Partially done work**: There is common ground between both taxonomies on reducing large batch sizes into smaller batches with an ideal of "continuous flow," where work is routinely moving through the system hence decreasing feature lead time. However, in Lean Software Development, *partially done work* is work that is not tested, implemented, integrated, documented, or deployed. Any feature description that is not implemented, any code that is not integrated or merged, any code that is untested, any code that is not self-documenting or documented, and any code that is not deployed where the user can receive value is *partially done work*.

The observed teams did not view materials flowing through the system as waste. Interaction designers need to produce just enough mockups, product managers need to write just enough stories, and developers need to write just enough code to make the story work. In any continuous flow system, there are unfinished materials at each step.

While we did observe a product manager starting too many features at once (as described in the *mismanaging the backlog* waste section), we mostly observed work flowing in a relatively orderly fashion with minimal delay. Large amounts of waiting designs or stories would have been classified as *mismanaging the backlog* waste.

Mismanaging the backlog includes observed wastes beyond *partially done work*, namely, sequencing low priority work before high priority work, accidentally duplicating work, capricious thrashing, or delaying necessary bug fixes. Lean Software Development does not cover these wastes.

Rework and **Defects**: While both taxonomies agree on *defects* as waste, our *rework* concept subsumes *defects*. *Rework* includes mistakes made by the product managers (in writing acceptance criteria), the interaction designers (in creating mockups) and the developers (in writing tests and code). Poor testing strategies and delaying testing can cause rework. *Rework* also includes shortcuts intentionally made by team members to save time leading to technical debt. Finally, our definition of *rework* distinguishes mistakes that could have been avoided from problems only obvious in hindsight.

Waiting/multitasking and **Task switching**: *Multitasking* and *task switching* describe the same behavior. The desire to have developers work on one thing at a time is common to both models.

Waiting/multitasking and **Delays**: In our model, *waiting* includes delays from not having the needed information or resources to get one's work done as well as the cost of slow tests and unreliable tests. In Lean Software Development, *delays* are "waiting for people to be available who are working in other areas" to provide needed information that is not available to the developers [46]. Both wastes describe the cost of missing needing information. *Waiting* subsumes *delays*.

Knowledge loss and **Relearning**: In our model, *knowledge loss* is the cost from members rolling off the team. In Lean Software Development, *relearning* is "rediscovering something we once knew" [46], which includes the cost of an individual not remembering a decision already made. The observed teams see forgetting as a natural part of the human experience, not as a waste. Included in *relearning* is failing to engage people in the development process. We did observe product managers having difficulty in involving some stakeholders, which we include in the *building the wrong feature or product* waste. *Knowledge loss* focuses *relearning* to simply regaining departed knowledge.

8.7 Results Evaluation and Quality Criteria

While other factors may affect software engineering waste, we focus only on those that we observed during the study. Grounded Theory studies can be evaluated using the following criteria [11, 63]:

Credibility: "Is there sufficient data to merit claims?" This study relies on two years and five months of participant-observation, 33 intensive open-ended interviews, and one year's worth of retrospections.

Originality: "Do the categories offer new insights?" This study is the first empirical study of waste in software development. It not only discovered new waste types but also supports and expands existing waste types that have not previously undergone rigorous empirical validation in a software engineering context.

Resonance: "Does the theory make sense to participants?" We presented the results to the organization. Seven participants reviewed the results and this chapter. They indicated that the waste taxonomy resonates with their experience: "These are pain points that we all felt. This covers what we learned on the projects" —Participant 33; "I have lived through these projects. . . No other waste comes to mind" —Participant 31. This process produced no significant changes, which is not surprising because participant observation mitigates resonance problems.

Usefulness: "Does the theory offer useful interpretations?" This study identifies software development wastes that are not identified in manufacturing and explains why certain behaviors, events, and actions can cause software engineering waste. It provides a rich waste taxonomy for identifying wastes in practice.

Regarding **external validity**, Grounded Theory is non-statistical, non-sampling research; therefore, the results cannot be statistically generalized to a population. Based on existing research and our experience, none of the identified wastes appear peculiar to Pivotal or Extreme Programming. However, Pivotal is a very effective organization, which uses iterative development and has been concerned with eliminating waste for almost two decades. Organizations that are newer, less experienced, less concerned with waste, or use less iterative methods may experience additional waste types. For example, *waiting* waste manifests in organizations that hand feature documents between teams or use large batch sizes of features [2, 33, 40]. Researchers and professionals should, therefore, take care to adapt our findings to their contexts case-by-case.

Finally, the results might be influenced by **researcher bias** or **prior knowledge bias**. During participant observation, the researcher may lose perspective and become biased by being a member of the team. That is, while a participant-observer gains perspective an outsider cannot, an outside observer might see something a participant observer will miss. Similarly, while prior knowledge helps the researcher interpret events and select lines of inquiry, prior knowledge may also blind the researcher to alternative explanations [21]. We mitigated these risks by recording interviews and having the second and third authors review the coding process.

8.8 Conclusion

This research presents the first evidence-based taxonomy of software engineering waste, identifies numerous causes of waste, and explores fundamental tensions related to specific waste types. Nine waste types are identified: building the wrong feature or product, mismanaging the backlog, rework, unnecessarily complex solutions, extraneous cognitive load, psychological distress, waiting/multitasking, knowledge loss, and ineffective communication. Each waste is illustrated with examples taken from observed projects at Pivotal, showing how the waste materializes, and in some cases how it is removed or eliminated. We also compare our taxonomy to Lean Software Development's waste taxonomy.

Our taxonomy emerged from a Constructivist Grounded Theory study, including the collection and analysis of data from two years and five months of participant-observation of eight software development projects; interviews of 33 software engineers, interaction designers, and product managers; as well as one year of retrospection topics. The analysis of the retrospection topics reveals that the observed Pivotal teams care deeply about finding and eliminating waste in their software development processes.

While this research supports parts of the Lean Software Development waste taxonomy, it differs in three key ways: 1) it introduces four new waste categories: *unnecessarily complex solutions, extraneous cognitive load, psychological distress,* and *ineffective communication*; 2) it does not support Lean Software Development's *handoffs* waste category; and 3) its waste categories are largely broader than Lean Software Development's categories. As such, our taxonomy is more expressive and more accurately describes the observed data. To be clear, the Lean Software Development's taxonomy of wastes was developed top-down, by mapping manufacturing wastes onto software development concepts. It was not empirically tested and therefore does not have the same epistemic status as our taxonomy, which was developed bottom-up from rigorous primary data collection and analysis.

Chapter 9

Conclusion

9.1 Summary

Using Grounded Theory, we examined the question, "what is happening at Pivotal when it comes to software development?" Listening to the concerns of the participants, the emergent research lead to three core concerns: 1) how do the observed Pivotal teams effectively construct software in a sustainable manner? 2) how do the observed teams engender team code ownership? and 3) what kinds of waste do the observed teams identify and remove in its journey of continuous improvement?

9.1.1 How does Pivotal construct software?

In exploring the construction of software, Grounded Theory guided us to the theory of Sustainable Software Development, a set of principles, policies, and practices used in the construction of software products so that teams survive team disruptions such as team churn. The principles include engendering a positive attitude toward team disruption, encouraging knowledge sharing and continuity, and caring about code quality. The policies are team code ownership, shared schedule, and avoid technical debt. The removing knowledge silos practices are continuous pair programming, overlapping pair rotation, and knowledge pollination. The caretaking the code practices are test-driven development / behavior-driven development and continuous refactoring which are supported by live on master.

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.

The theory is rooted in team code ownership, as removing knowledge silos and caretaking the code enable teams to be able to modify any part of the code base.

9.1.2 How does Pivotal engender team code ownership?

In researching how Pivotal engenders team code ownership, the observations clearly indicate that team code ownership is a feeling to be engendered not a policy to be decreed.

Meanwhile, both discussions with and observations of participants suggest five factors associated with strong feelings of team code ownership. Pivotal developers more acutely feel team code ownership when i) they understand the system context; ii) they have contributed to the code in question; iii) they perceive code quality as high; iv) they believe the product will satisfy user needs; and v) they perceive team cohesion as high.

Moreover, diverse events and trends can undermine sense of ownership, including: increasing knowledge silos, increasing code base size, increasing team size, inability to contribute, pressure to deliver and deprioritizing continuous refactoring, ignoring user feedback, ignoring developer feedback, and distancing a developer from the team.

In reviewing the data, we discovered that transitioning from individual code ownership to team code ownership is not easy for some engineers as psychological ownership fulfills deep psychological needs.

9.1.3 What kinds of waste does Pivotal identify and remove in its journey of continuous improvement?

In investigating removing waste, we created a waste taxonomy that identifies the type of waste the participants detect and try to remove in the studied organization. The wastes are building the wrong product or feature, mismanaging the backlog, unnecessary complexity, rework, unnecessary cognitive effort, cognitive hindrance, waiting, and ineffective communication. Contrary to the Lean Software Development's taxonomy of wastes, which is top-down and created by mapping manufacturing wastes to software wastes, our taxonomy is bottom-up as it is grounded in empirical data. The comparison of the two models shows some alignment. However, our taxonomy expands the Lean Software Development's taxonomy by broadening the definition of most wastes and introducing additional wastes not previously identified. As such, our taxonomy is more expressive and more accurately describes the observed data.

As a model, the waste taxonomy succinctly describes waste types and is more descriptive than previous work.

9.2 Future Research

Given the emergence of the theory of Sustainable Software Development, a model of team code ownership, a waste taxonomy, and a deeper understanding of the evolution of Extreme Programming, there are several opportunities for future research.

The theory shows how the principles, policies and practices work together to achieve the business goal of sustainability. It would be interesting to understand how variations of the principles, policies and practices might influence the overall process. For example, what would be the impact of removing or altering the shared schedule? What would be the impact of not doing test driven development? Additional experimentation can be done to assess each practice's flexibility while still maintaining the ability for a team to survive disruptions.

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. Future work could 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. It would be interesting to 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.

At Pivotal, the waste taxonomy could be used to systematically examine potential wastes. Since some wastes are rarely salient unless one is specifically looking for them, future research could use the waste taxonomy to see how helpful it is in identifying these waste on a project. We could investigate how Pivotal relies on feedback loops as a mechanism for identifying, dealing with, and reducing waste. Also, the waste taxonomy could be verified at additional companies using different development methodologies. It is possible that a team following scrum or waterfall might generate waste not encountered on the observed teams.

Pivotal has evolved the Extreme Programming practices for building and managing a backlog. Instead of committing to work to be done each week, teams work off the top of the backlog in an iteration-less flow. Additional research could examine how the evolution of the Extreme Programming backlog practices affects software development.

It would be interesting to validate the emergent theories at other organizations and compare the results, as this would lead to a deeper understanding of software development. Now that extensive field work and Grounded Theory has produced these theories, additional research could validate each theory in a very different context.

9.3 Initial Impact at Pivotal

I have presented this research at the Santa Monica, Palo Alto, San Francisco, and Toronto Pivotal offices. Each presentation results in lengthy conversations about the implications of applying the research at Pivotal and discussions about people's experiences affirming the research results.

As a result of these presentations and paper publication [56, 57, 58], many teams are now conscientious about their selection of pair rotation strategies. Before presenting the research results, common pair rotation strategies included random, ad hoc, and optimizing for people rotation. Teams now consider which pair rotation strategy will result in increasing knowledge sharing among team members.

Many teams now realize the hazards of knowledge silos. If a team does decide to have someone continue with a track of work, the team acknowledges the risk and considers the costs versus benefits.

Some teams acknowledge the difficulty for new hires in switching from individual code ownership to team code ownership. Coaching individuals through this process may lessen the feelings involved in altering psychological ownership.

One team in the Denver office is using the software development waste taxonomy as a structure for their weekly retrospections. Reflecting on the waste types helps the team systematically review their software development process and reveals topics that the team does not normally discuss.

Two software developers in the Munich office listed out common software development issues on projects. They then successfully mapped these issues into the software development waste taxonomy. Their work confirmed that the waste taxonomy covers issues and wastes observed at Pivotal.

Finally, one software developer provided this feedback after reading the paper on software development waste taxonomy:

"The analysis made me feel validated above all else... I have felt the pain of software development waste, but did not feel justified in expressing the negative effects to the full vigor in which I felt it. This is because I felt the waste causes seemed petty and small when thought about individually, but as I read your paper, I learned that the causes I felt fed into entire categories of waste. Now those causes are not petty anymore in my mind. I feel empowered to speak about them more often in retros because they are just symptoms of bigger problems - software waste."

9.4 Conclusion

Pivotal's continuous improvement practices have resulted in an evolution of Extreme Programming. Each project becomes an experimentation opportunity for the engineers to try new ideas and see the results. As compared to a company with a relatively stable product line, Pivotal Labs has a unique opportunity for continued experimentation as it undertakes scores of projects each year. One Pivotal engineer reflected that *"each project feels like a hill climbing optimization algorithm, exploring the feature set of the product. Each week, stories guide the product towards a delightful user experience."* With each new project, the teams have an opportunity to refine Extreme Programming in ways not envisioned by its creator.

Appendix A

Waste Examples and Chain of Evidence

This section lists out the chain of evidence for the software engineering waste taxonomy. This emergent theory comes from participant observation and analysis of retrospection data. For each listed retro item, participant observation confirmed the issue. In a few instances, participant observation provided examples not identified in a retro.

Waste Category: Building the wrong feature or product

- **Cause Category**: User Desiderata
  - *Cause Property*: Not doing user research, validation, or testing
    - Retro Topic: Not getting access to actual customers
    - Retro Topic: Unable to find users to interview
    - Retro Topic: Not thinking about the user
    - Retro Topic: No testing for a few weeks
    - Retro Topic: Implementing features not validated by users
    - Retro Topic: Not doing usability testing
  - *Cause Property*: Ignoring user feedback
    - Retro Topic: Ignoring user feedback
    - Retro Topic: Ignoring user research
    - Retro Topic: Building a feature that users do not want
    - Retro Topic: Redoing mock-ups because stakeholder chooses to ignore user research
  - *Cause Property*: Working on low user value features
    - Retro Topic: Unable to articulate the value to the user
    - Retro Topic: Building unnecessary features
    - Retro Topic: Working on high effort features for low user value

Retro Topic: Creating demos for stakeholders

Retro Topic: Wanting to build every feature that comes to mind

Retro Topic: Creating mockups for features needed in the distant future, not the present

Retro Topic: Load testing site with expected low usage

Cause Category: Business Desiderata

Cause Property: Not involving a stakeholder

Retro Topic: Not closing the loop with a stakeholder

Retro Topic: Disconnect between team and stakeholders

Retro Topic: Disconnect between team and headquarters

Retro Topic: Not enough stakeholder involvement

Retro Topic: Missing product owner

Retro Topic: Not knowing the product owner's goals

Retro Topic: Decision makers elsewhere

Cause Property: Slow stakeholder feedback

Retro Topic: Slow feedback from stakeholder

Retro Topic: Wanting more involvement from the product manager

Retro Topic: Slow turn around to request for feedback

Cause Property: Unclear product priorities

Retro Topic: No vision for next scope of work

Retro Topic: Lack of focus

Retro Topic: Unfocused scope

Retro Topic: Unclear priorities

Retro Topic: Inconsistent priorities and directions

Retro Topic: Unable to sequence or prioritize the work

Waste Category: Mismanaging the backlog

Cause Category: Backlog inversion

Retro Topic: Most important items are not at the top of the backlog

Retro Topic: Difficulty in finding valuable work

Retro Topic: Starting stories that are not at the top of the backlog

Cause Category: Duplicated work

Retro Topic: Duplicated story in the backlog

Retro Topic: Forgetting to start a story

Retro Topic: Pushing code to a branch that is forgotten and reimplemented by team

Retro Topic: Two pairs doing the same needed work without creating a chore

Retro Topic: Not looking at stories in flight to know who is doing what

Cause Category: Starting too many features

Retro Topic: At release turning off feature flags for partially completed work

Cause Category: Not enough ready stories

Retro Topic: Story desert

Retro Topic: Shallow backlog

Retro Topic: Not enough tracks of work

Participant Observation: Engineers writing feature stories

Retro Topic: Ad hoc stories

Retro Topic: Many blocked stories

Cause Category: Imbalance of feature work and bug fixing

Retro Topic: Delivering only bugs and chores this week

Retro Topic: Two weeks of features, two weeks of bugs, repeat

Cause Category: Delaying testing or critical bug fixing

Participant Observation: Prioritizing feature work over testing

Retro Topic: Delaying end-to-end testing

Retro Topic: Not looking for defects until end of a release

Retro Topic: Critical bugs not prioritized when detected

Retro Topic: Working on features for two weeks, working on bugs for two weeks

Cause Category: Capricious Thrashing

Retro Topic: Fiddling with the sequence and number of steps in the user registration process

Retro Topic: Repeatedly re-sequencing the order customization process

Participant Observation: Engineers prioritizing stories

Waste Category: Unnecessarily complex solutions

Cause Category: Unnecessary feature complexity from the user's perspective

Retro Topic: Design and development not discussing upcoming work

Retro Topic: Mockup is too complex and could be simplified

Retro Topic: Business logic is too complex and could be simplified

Retro Topic: Too many edge cases

Participant Observation: Feature does not solve problem in simpliest way

Participant Observation: Cheaper engineering solution to the "sample" problem

Cause Category: Unnecessary Technical Complexity

Retro Topic: Too much context needed to understand system

Retro Topic: Datacenter architecture could be simplified to achieve same goals

Cause Property: Duplicating code

Participant Observation: Two classes do the same thing

Participant Observation: Code logic is not in one place, missing cohesion

Participant Observation: Competing solutions for test setup

Cause Property: Lack of interaction design reuse

Retro Topic: Every page is unique

Retro Topic: Page specific stlying means no css reuse

Retro Topic: Team following styleguide-driven-development with limited designer buy-in

Retro Topic: Inconsistent look and feel

Cause Property: Overly complex technical design created up-front

Retro Topic: Changing up-front design because of new information

Retro Topic: Created design is too complex for problem

Retro Topic: By over designing, system re-invented the wheel

Retro Topic: Big design up-front feels like busywork

Waste Category: Extraneous cognitive load

Cause Category: Complex or large stories

Retro Topic: Lots of edge cases

Retro Topic: Mananging too much complexity in one super large story

Retro Topic: Repeated rejection of super large story

Interview: Large stories with lots of comments

Cause Category: Overload from context switching

Participant Observation: Jumping between too many tasks: “I am so confused. What were the details about this task? I do not remember.”

Participant Observation: Overwhelmed with state: “What problem are we trying to solve again? I need a break.”

Cause Category: Inefficient tools and problematic APIs, libraries, and frameworks

Retro Topic: « many different tools »

Retro Topic: « many different APIs »

Retro Topic: « many different libraries »

Retro Topic: « many different frameworks »

Cause Category: Suffering from technical debt

Retro Topic: Code smells

Retro Topic: Creating too many chores than the team can accomplish to deal with technical debt

Retro Topic: Refactoring needed "everywhere"

Interview: "Siteminder makes me angry enough that I want to hack into it, expose how useless and horrible it is and wipe this miserable product off the face of the earth!"

Cause Property: Hard to change code

Retro Topic: Code is getting harder to change

Retro Topic: Lots of coupling and dependencies between classes

Retro Topic: High code complexity

Retro Topic: Code base becoming unmaintainable

Retro Topic: Copy and pasting instead of refactoring

Retro Topic: State is everywhere

Retro Topic: Code that does not play to strength of programming language

Retro Topic: Lacking confidence to change code

Cause Category: Inefficient development flow

Retro Topic: Git force push (overrides version control system)

Retro Topic: Not running test before delivering code

Retro Topic: Commits that break the build

Retro Topic: Merge conflicts due to long lived branches or large changes

Retro Topic: Delivering code that is not done

Retro Topic: Broken build

Retro Topic: Pushing code on a broke build

Retro Topic: Continuous integration is red this week

Retro Topic: Deploying software is painful

Cause Category: Sluething information

Retro Topic: Unclear bug reports

Retro Topic: Unclear commit messages

Retro Topic: Reverse engineering technical decisions

Retro Topic: Competing code patterns

Participant Observation: Messy code

Cause Category: Noisy output from tools

Retro Topic: Distracted by useless output when running tests

Waste Category: Psychological Distress

Cause Category: Low team morale

Retro Topic: Frustrated developers

Retro Topic: Not managing expectations

Retro Topic: Negative attitudes

Retro Topic: Apathy

Retro Topic: Not knowing everyone on the team

Retro Topic: Project feels like it is falling apart emotionally

Retro Topic: Unacknowledged by management

Retro Topic: Messy code decreasing sense of ownership

Retro Topic: Imbalance of bugs to feature work

Retro Topic: Poor lighting, lack of windows

Cause Category: Rush mode

Retro Topic: Fixed set of features with a fixed timeline

Retro Topic: Not knowing features for the release

Retro Topic: Aggressive timelines

Retro Topic: Shifting deadline

Retro Topic: Scope creep

Retro Topic: Prolonged rush mode

Retro Topic: Unrealistic expectations

Retro Topic: Repeatedly hearing "this has to be done today"

Retro Topic: Long days

Retro Topic: Overtime

Cause Category: Interpersonal or team conflict

Retro Topic: Criticizing in public

Retro Topic: Difficult pairings

Retro Topic: Not listening

Retro Topic: Pairing fatigue

Retro Topic: Interpersonal conflict

Waste Category: Rework

Cause Category: Technical Debt

Retro Topic: Technical debt

Retro Topic: Delayed refactoring

Retro Topic: Encouraging team to keep on refactoring

Retro Topic: Limits on refactoring

Retro Topic: Delaying refactoring until when convenient

Retro Topic: Rushing to fix "bugs" instead of delivering quality at a steady pace

Retro Topic: Rushing to deliver code creating technical debt

Retro Topic: Copy and pasting code makes future code development harder

Cause Category: Rejected Stories

Retro Topic: Rejected stories

Cause Category: No clear definition of done

Cause Property: Ambiguous story

Retro Topic: Unclear acceptance criteria

Retro Topic: Poor acceptance criteria leading to stories getting accepted that do not work

Retro Topic: Bug reports missing repeatable steps

Retro Topic: Rushed stories in the backlog

Cause Property: Second guessing design mocks

Retro Topic: Second guessing design mocks

Cause Category: Defects and Bugs

Cause Property: Defects and Bugs

Retro Topic: Broken feature

Retro Topic: Application is in an unshippable state

Retro Topic: Adding fake pages or code makes system unshippable

Retro Topic: Finishing stories before delivering

Cause Property: Poor testing strategy

Retro Topic: Green build, broken code

Retro Topic: Sparsely tested component

Retro Topic: Not testing code, untested code

Retro Topic: Increasing number of bugs

Retro Topic: No test failure resulting in rework

Retro Topic: Hurrying to get stories finished resulting in more defects

Cause Property: No root-cause analysis on bugs

Retro Topic: Not learning from defects and team's mistakes

Retro Topic: Not understanding what is root cause of tech problems

Waste Category: Waiting/multitasking

Cause Category: Slow tests or unreliable tests

Retro Topic: Slow tests

Retro Topic: Flaky tests

Retro Topic: Randomly failing tests

Cause Category: Unreliable acceptance environment

Retro Topic: Acceptance server is unreliable

Cause Category: Missing information, people, or equipment

Retro Topic: Missing equipment

Retro Topic: Missing product

Retro Topic: Missing computer

Retro Topic: Missing headsets

Retro Topic: Missing information from stories

Cause Category: Context Switching from delayed feedback

Cause Property: Product manager taking a long time to accept or reject a story

Retro Topic: Many stories waiting for acceptance

Retro Topic: Long stories acceptance feedback loop

Retro Topic: Builds going out with unaccepted stories

Retro Topic: Hard to figure out why a story is rejected

Cause Property: Unavailable product manager or interaction designer

Retro Topic: Missing product manager

Retro Topic: Missing designer

Retro Topic: Not quickly responding to questions

Retro Topic: Product has a lot on their plate

Retro Topic: Discussing about getting more product managers

Retro Topic: Designers are missing in pairs

Retro Topic: Not enough designer/developer pairing happening

Waste Category: Knowledge loss

Retro Topic: Knowledge loss from team churn

Retro Topic: Team member departure

Retro Topic: Knowledge silos forming

Waste Category: Ineffective communication

Cause Category: Team size is too large

Retro Topic: Improving communication across the whole team

Retro Topic: Improving collaboration between teams

Retro Topic: Entire team not in discussion about changes

Retro Topic: Coordination problems since team is too large

Retro Topic: Difficult retros since team is so large

Cause Category: Asynchronous communication

Cause Property: Distributed teams

Retro Topic: No communication channel for remote meeting attendees

Retro Topic: Distributed teams

Retro Topic: Too many asychcronous communications through slack

Retro Topic: Remote pairing is harder

Retro Topic: Distributed retros

Retro Topic: Split across timezones

Retro Topic: Long communication threads in stories in backlog

Cause Property: Non-collocated stakeholder

Retro Topic: Non-collocated stakeholder

Cause Property: Dependency on another team

Retro Topic: Not communicating changes when making breaking changes

Retro Topic: Coordinating changes in APIs

Retro Topic: Blocked on other teams

Cause Property: Opaque processes outside team

Retro Topic: Opaque processes in other teams

Cause Category: Imbalance

Retro Topic: Resistance to quick team huddles

Cause Property: Dominating the conversation

Retro Topic: not including people in the conversation

Retro Topic: two people dominating meetings

Retro Topic: quieter voices needing to speak up more

Cause Property: Not listening

Retro Topic: Not enough listening

Cause Category: Inefficient meetings

Cause Property: Lack of focus

Retro Topic: Inefficient planning meeting

Retro Topic: Meetings turning into long intense discussions with no resolution

Retro Topic: Not reviewing action items

Cause Property: Skipping retros

Retro Topic: Skipping retros

Cause Property: Not discussing blockers each day

Retro Topic: Not communicating critical status updates

Cause Property: Meetings running over (e.g. long standups)

Retro Topic: 35 minute standups

Retro Topic: Long standups

Appendix B

Interview Drawings and Partial Transcriptions

I asked many of the interviewees an open ended drawing question to begin a conversation by eliciting the their perspective on a topic. Here I include the diagrams with an abridged interview transcript where the interviewee describes their drawing.

B.1 Interview 1 Abridged Transcript with Product Manager

Todd: This is a drawing exercise for you. So, pretty open-ended question, could you describe a project work flow by drawing it on that sheet of paper? There's no wrong answers. 02:32

Interviewee: Does it matter if there's a DNF first or...? 02:39

Todd: However you want it. Typical, ideal, however you want to draw it. 02:44

Interviewee: We actually show this to clients. We have this drawing here. 02:54

Interviewee: Most of the projects that I've been on start with the DNF. We'll call this DNF. 03:07

Interviewee: It's assuming the project has design in development going. This kind of, like, inception. 03:27

Interviewee: And this is dev inception. 03:32

Interviewee: This is your discovery, and framing. 03:50

Interviewee: Here in this area, we're sort of doing more exploratory research and sort of going wide trying to understand the users. That's something we do at the beginning of every project especially if the client doesn't know who their user is or they have ideas of who they are but it's not validated. 04:10

Todd: That's the discovery phase. 04:11

Interviewee: Yes. So, this is like exploratory user research. That's like interviews and maybe on site, shadowing people and things like that. And once we have a good idea of who they are, we start wire

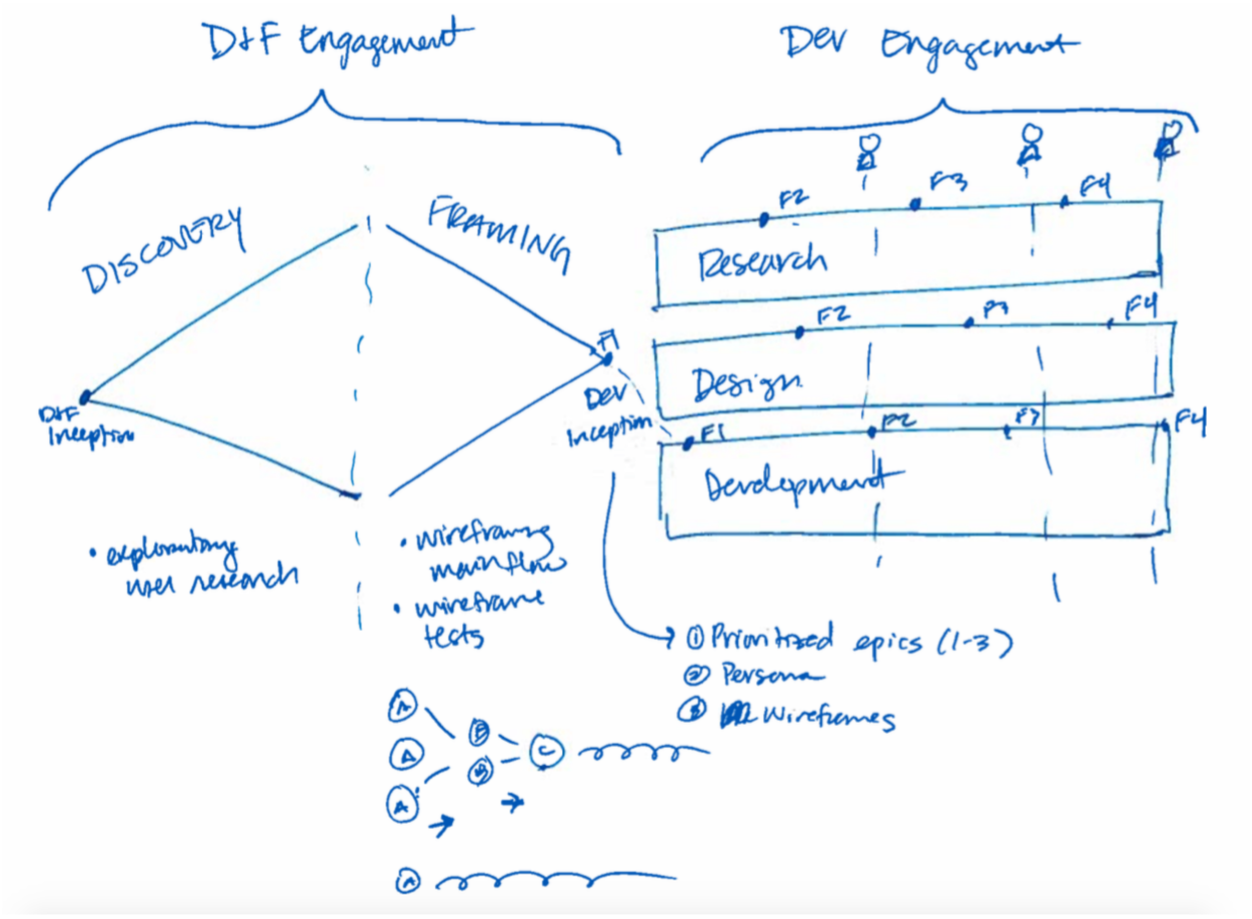


Figure B.1: "2015-05-29's drawing of a project work flow"

framing, maybe like the main flow and doing some wire frame tests like user research, that kind of thing and that's kind of within that, I think, we're kind of iterating as well. 04:45

Interviewee: I guess you can either start with one and iterate on that or you can start with many and narrow down. We've done both. So, when you start with many, you just- So, this is like your different As, research, then research to get to C. That makes sense? 05:10

Todd: It's like a portfolio of ideas? 05:14

Interviewee: Yes. I did it on my first project and it worked out really well and then from there you're kind of iterating on what you've come up with. 05:25

Interviewee: We came up with a couple, different nuance ways of doing something because they all seem good and they're like, two or three different features and we sort of combined them in different ways and then we took our insights from that and narrowed it down two versions, narrowed it down to one version. 05:46

Interviewee: I've also done it where you just start with A and you just iterate. I think both worked but this is fun to do. By the time we get to dev inception, we have prioritized epics. You have a develop persona. And you have wire frames. That's kind of the ideal for this point. At that point, development can start on say feature one and in the meantime, we'll start researching feature two. Then, we'll develop it. Then, feature three. So, it kind of goes like that. 06:37

Todd: Nice. 06:39

Interviewee: Maybe you have feature one here and then we'll go here. While they're doing that, we'll start on the next thing. So, on the next thing it flows down. I hate saying waterfall because I know it's the wrong kind. It's like a different kind of waterfall but it iterates in that way. So, we're like, "Oh, it's going in the cycle." At any time, we're going to bring in users to test whatever we want. 07:09

Todd: You're drawing people. 07:12

Interviewee: Yes. These are users. At any given point, we can even do exploratory research. We can do wire frame or visual research or we can show them an actual prototype. It's kind of nice because you can, whenever you need people, whenever you're stuck on something, you don't have to wait for the right time to bring people in. It's just you can test anything at any given time. 07:34

B.2 Interview 6 Abridged Transcript with Anchor

Todd: So on this sheet of paper, could you draw your perspective on our process for software development. 0:00:09

Interviewee: Our process for software development. Where do you want to begin? Or is that like . . . 0:00:14

Todd: It's a really open-ended question. 0:00:15

Interviewee: So there's a customer, a potential customer. And they come and there's some kind of vetting to, at least, get them in the door. We think that they have interesting idea of some kind, and maybe we ask a couple of questions; so this might be the OD or sales. Just some minimal vetting and then just enough for them to say: "you know what, I think we could possibly have a conversation about what it is that you want to have built." So then we do a scoping with them, and in that scoping, the input is their general idea. Usually, it's like pretty vague; It could be from very vague to they totally know what they're talking about both on their problem space and with the solution to look like. So then the output of that is a document that basically tells them this is what it likely is, and this is how much it's gonna cost, roughly speaking, and these are gonna be the roles and responsibilities, this is what you would be buying. What just really actually, some amount of hours of some amount of people, different skill, that's what we're

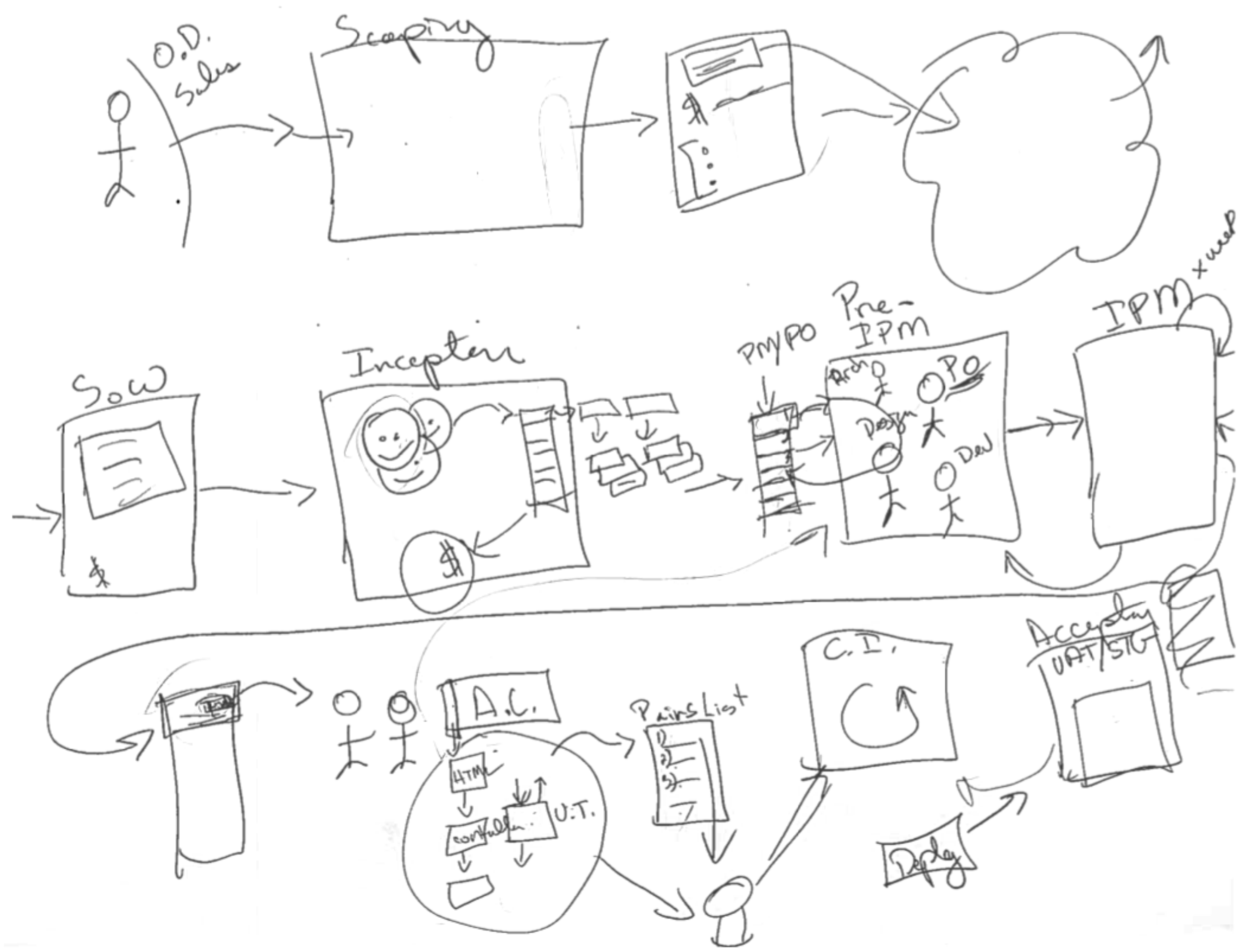


Figure B.2: "2015-08-12 Anchor's drawing of software development process"

actually promising and we'll aim for this thing, but we'll always constantly trying to give you the best value thing as we go along. So that's some kind of proposal, I don't know exactly what we call that, but it is all part of the scoping. 0:02:00

Interviewee: And then, I'm gonna draw, like, maybe a number of stops here; by the time I see it, somebody signed, like, an SOW or similar that was possibly, like, but influenced by that scoping value but who knows what negotiations or whatever happened after that; and probably informed by that document, like, what we say we're aiming to deliver. But this is, I think, this is more like a legal document or as the SOW is more like a legal document whereas the thing that came out of scoping is more of like a "come work with us" and some of the other stuff around that. So then, we go into an all of these assumes it's ok. I'm talking happy path. Cool? 0:02:59

Todd: Yeah, I'm good. 0:03:00

Interviewee: So then we do an inception, and this is where, regardless of what we said over here, we get them to get really clear about who is the person that they're targeting; like, ok so it's about a product market fit, who's the market, what's the product gonna be. So here's this person or persons or personas, and then for each of one those, like, what are the kinds of things that they need to get done, right? So they've got things that need to happen, that ultimately will result in to some kind of outcomes, and it almost always like it's gonna convert to dollars in some way, shape, or form. Sometimes it's a non-profit thing and that's slightly different, there's mixed motivation. So then what we do is we say, "ok, in order to meet those tasks, what we need is features from the product" and so we're calling out, like, epics, if you will, feature sets, whatever, and then when we start breaking those down into individual stories and these are just sketches at this point. We're not gonna get all into acceptance criteria right away, it's just like trying to enumerate coz really the outcome of the inception is a backlog. 0:04:43

Interviewee: So you've got some product owner, who culls the outcome of that into a prioritized list of stories, each one of those describes at tiny interaction between this person and the software that we're building. Ideally, there's variations on that but then, so that's what an individual user story is. 0:04:58

Interviewee: And then we go to kick off. So then, we have our first iteration planning meeting where we take as input the prioritized backlog and for each story we go through them. And by then, the product owner has got them into a point where I call it readied, they've met the definition of ready, which is they have clear acceptance criteria. There might even be, a pre-IPM where that work is done, as well as, so this is the input is the backlog and the output is the backlog in the pre-IPM. There are two things that happen, one is that we get clarity early on what the requirement is, and the other is that we get technical input to help with that prioritization and viability, like, "ok this may seem simple, like on the face of it", but actually, there's a whole of things that has to happen to make that work, etc. So we surface some of that, the pre-IPM includes somebody from the product side of the house and someone from development, so product owner, development. Sometimes depending on those, the story can get more complicated, the more requirements, with the greater the variety or the more exotic the product is. So you might even need someone in from design who is kind of help guiding how this should unfold and the interactions between things because there's all kinds of decisions from that end. I Imagine, although, we haven't done this yet, in an enterprise client that you might even have somebody from like architecture there, business architecture. The point of this conversation is to get all of the perspectives that need to be folded into the prioritization and the validation of, that the stories are legit. 0:07:11

Todd: Yes. 0:07:12

Interviewee: So then that goes into IPM, and this is a straight-forward roll through - the from top to bottom - the backlog, and we go over each story. We read out title, we read through the acceptance criteria and then the team points each of the stories, the purpose of that is to surface complexity and to get a general, like, understanding, like, common understanding of what this thing actually is, what it is, what's gonna be involved to building it. 0:07:52

Todd: Yes. 0:07:53

Interviewee: We don't give in to like, we try not to give in to implementation details. Sometimes, we have to dip down for a second to like verify that we're all speaking the same language, that we're all really envisioning the same kind of thing. But that usually is surfaced by like, "I pointed at one and you pointed at five," and then I would like "what?" So then that hopefully prompts a conversation. If we all said three, it's good enough for now we move on, if we all think it's about the same thing and the product owner doesn't lose their mind hearing that number, then we're all kind of okay. Then the output of the IPM is individual points on the stories, and we typically go for some amount of horizon so the minimum that I feel comfortable walking out with is at least for the week, so we don't have to, like, break the flow of getting work done before the next IPM because we're gonna do this once a week, these IPMs we're gonna do this once every week. But ideally, a little bit more runway so that we mitigate against that the team haven't jumped out of the flow and also to, like, help the product owner have some room to sort of steer. If they only have so many points in the stories, then they have to kind of pay the price of reprioritizing. So that's IPM. Haven't have written a line of code yet. So then, after IPM, we get working. So out of the backlog, a pair picks up a story and so then that pair looks at the story... How deep do you want me to go, because I can go all the way down to like testing and stuff. 0:09:48

Todd: Sure. 0:09:49

Interviewee: Ok. 0:09:50

Todd: This is your diagram. There's no wrong answer. 0:09:54

Interviewee: Alright. Ok. So, the pair looks at the acceptance criteria and they say, "hmmm.... what test do I need to write that will help? Or a set of test that will help if those tests ran green? I feel really confident that we've met the spirit of that story." So they start there. Usually at pivotal will do typically outside in so that means if we write something that looks like an acceptance test, something that describes very closely what this person with the persona experiences both in what they do, and what they see back from the software. So we write the starts, we start with a very low fidelity version of that. It's not gonna describe the whole interaction, it might describe the smallest piece of we could possibly articulate; so we start there. And then we run that test and it fails. Then we say, "hmmm, ok, so we're in this architecture, what's the next part that we need to build in order to begin to meet those needs?" And we work our way down the architecture. In a typical web app, there's something that's displaying, an HTML page, and then there's something that is probably orchestrating the generation of that HTML, like a controller and usually we try and separate our concerns. We think about these things as we work our way down driven by trying to meet just this one acceptance test. 0:11:26

Interviewee: So along the way what we do is we write individual unit test for the components that have interesting behavior. The controller does have interesting behavior. It takes in some input and it makes some decision about what should be the output, what should be the resulting HTML. And that controller also interacts with other collaborators. So we have tests that say, are you properly handing off these parameters? So these really fine-grain unit tests. But the key is that these things are, each time we write these tests, we're setting an expectation on that little tiny piece of the system, in the same way that our acceptance criteria are setting an expectation on the software, and our user stories are setting expectations on the feature . So forth, all the way back up. We're trying to be needs-based all the way down as we do this; that's probably good enough. The details of how that happens varies wildly and even like within this, there are different schools of thought about how that happens. There's people who believe in writing units for every little thing and people who say "no, you can set certain bullwarks" and write tests around the bullwarks. Let everything sort of float in between. 0:12:43

B.3 Interview 7 Abridged Transcript with Product Manager

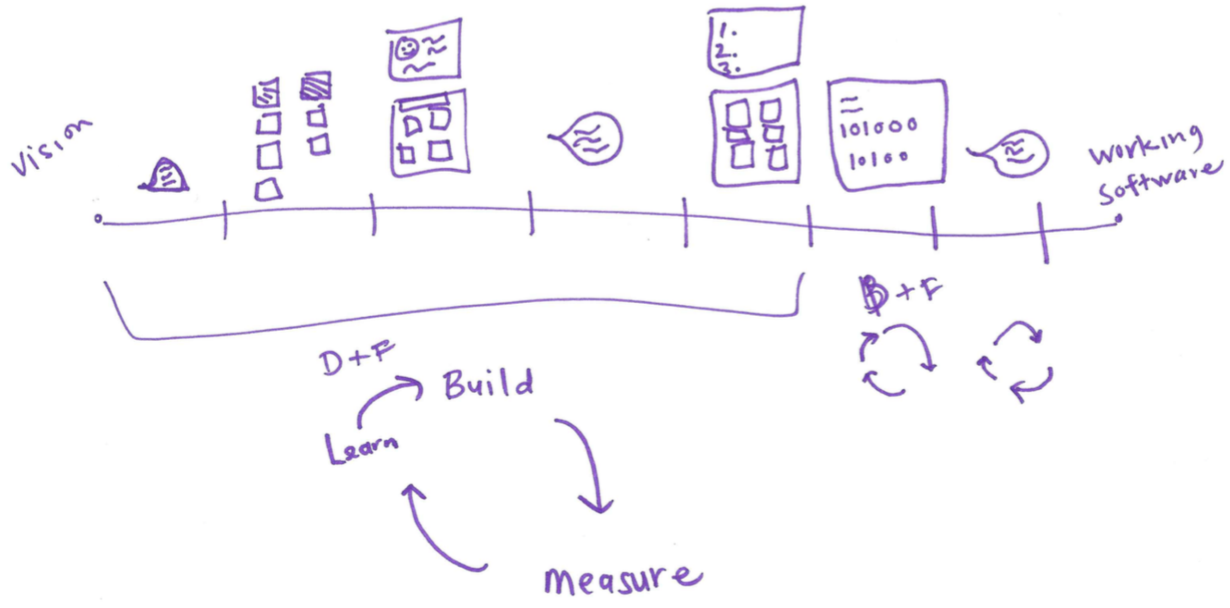


Figure B.3: "2015-08-12 Product Manager's drawing of software development process"

Todd: If you're open to it, could you, on that sheet of paper, draw out how you view the way we build software? This is completely open-ended. There's no wrong answers. And just take a stab at it. 0:00:21

Interviewee: So I'm drawing a line, it's like product on a continuum. We're gonna have vision here. We're gonna have working. Can you tell I'm a PM because I'm writing in lines, boxes can't do? 0:00:45

Todd: I love it. 0:00:46

Interviewee: So let's see here. So let's do a couple of lines here. Alright, so let's say, each of these represents two weeks. So we'll say that the first point, so how we build it at pivotal is that what you asked? 0:01:17

Todd: Yes. 0:01:18

Interviewee: So it starts with having clients. 0:01:20

Todd: How would you build it too? 0:01:21

Interviewee: Yeah, yeah. Totally. The reason I asked about Pivotal is 'cause we're doing consulting. So if you're thinking of product as a continuum, our clients come in in all these different ways of where they the insert. But it starts with a conversation, kind of like a qualifying conversation of like, "hey, what do you wanna build?", and "what's the fidelity of your idea?" And from there, we have a good understanding of if they're at a stage where they can build something, or if they maybe really need to think about it further. But once we realize ok, they're ready for Pivotal , we'll start with a Discovery and Framing and not every project gets Discovery and Framings, but in the LA office, a lot of them do. We're trying to get most projects getting some a semblance of Discovery and Framing. 0:02:04

Todd: When would you not do one? 0:02:07

Interviewee: So we won't do one... I think we can make the case really for all projects could do with a little bit of DNF right? So even if you're like, I know where to build from. So with FYI rather, but for a PM, I'm thinking, okay my role is to help the client understand who are we building for and then what we are building and when? So for the, who are we building for, I think all clients. It'd be great if we could do some user research with them even if they are like, we did all these user research just to do a quick gut check, like hey this makes sense, that'll be great. 0:02:42

Interviewee: But I think a lot of the times when we don't do them, it tends to be convincing the client or if there's a budget concern. So they have enough of a fidelity of understanding who their target user is and who the secondary users are and have a persona and are focused on stuff like kinda key factors, ok, if you're saying you want to build an app for the millennials and for college-age students and for the ages 25-35, that's pretty vague. We do get that a lot, but we're saying, "ok, it's for millennials, so what gender are we going for?" Or is it, "what are their behaviors like?" There's a lot of questions that we can ask on this conversation. and these qualifying calls to say, to kind of fish out, to think if they are ready to work with Pivotal, or if they have enough information for DNF or not. So, does that answer? I guess that was a very specific answer. 0:03:44

Todd: That was great. 0:03:46

Todd: I interrupted your flow. 0:03:48

Interviewee: No, no, this is fine. So I'm of the opinion that I would love it if we could have some sort of research with all of our clients and users. Typically when we do a DNF, Discovery and Framing, we do that for 4 weeks on average, sometimes it's up to 6 weeks depending on how many users they have and how many people they want us to focus on. But we typically really try to work on 2 to 3 target users being 1 primary user and maybe 2 users of the system that kind of insert in their day. And then we've done a 2 week of design first or Discovery and Framing but really it's just "let's talk to some users and validate some ideas." But the whole goal of this Discovery and Framing process is to do a couple of weeks of talking to users, of about 2 weeks and that's when we're doing some of those exploratory interviews, kinda turn it into elicit narratives to understand what their behaviors and what their days are like. 0:04:51

Interviewee: And then from there, we can isolate what are some of their pain points and what are some of the frictions and inefficiencies and how are they capturing data and what are the tools that they use and who are the people they're talking to. From there, I guess one thing I didn't mention which is important is to say, what are the product goals that our clients have? And what are some of the assumptions that they have about their products? About their users where their products can help solve their needs. We go into user research, alright, here are some assumptions, hypothesis that we have, let's test them. Out of that output of user research is that we have this, we do some synthesis and analysis of all of the things they're talking about. We record the things that we saw, so if they're in a cubicle and they have tons of printed out papers because what they do is they get stuff by mail and then they have to scan it in. Those are the things that we're seeing that are part of their day that can affect them. 0:05:54

Interviewee: What did we hear, like things that they're telling us about their day, and things like we felt that they're telling us. But maybe there are some subtext and some nuance there saying everything's great but their faces are really strained and you can tell they're really frustrated and their posture has changed when they're talking about certain subjects. Kind of taking all of the things that recording from our user sessions and then coding it by what we saw, what we heard, what we felt and then finding themes. What are users talking about? Usually when we do research, we try to do 3 to 5 users, so that way we have a good cross-sample and in case there's any extreme people that we meet, it kind of helps us give a better analysis, better data sample. So we'll kind of call all the information that we have and we'll go to them, too. So we'll go to their cubicles or wherever their workspaces are. So we want to get a sense of their environment. Cause there's so much contextual information there that you can't get just from having a phone call with someone. 0:07:02

Todd: Yes. 0:07:03

Interviewee: So, then, that's usually about a couple of weeks doing some researching. As we're doing the researching, we're capturing all of our information on notepads and then we're doing what we call infinity mapping, affinity mapping. We'll get one of those big foam cork white boards and we'll take a listen to the recordings of when we're doing user research or if we can't record just looking at our notes because our notes are like, you're almost writing verbatim what people are saying. Because you don't want to put all your analysis in there at this point, you just want to get them to talk and get it all in, and then we'll take each kind of idea and we'll put it on a post-it note and we'll have a bunch of post-it notes around and they'll be coded by what we saw, what we heard, what we felt, and then we'll start seeing themes around this post-it notes. 0:07:54

Interviewee: So there's this one user that said there is a lot of things around education and tools and timeline and whatever it is. Then we started noticing trends, so we'll start taking those nuggets and we put them across this themes and then we'll do that to all our users; and then from there, we'll do another round of synthesis and the ideas are going to keep condensing. So then we have a synthesis of all the people we talk to and what the overlaps are and the themes of what their day is like, and the behaviors they drawn during that day. And then after that, we'll map out their day, like what are the tasks that they're doing, and then map out from the tasks all those insights rather not insights but nuggets of what we heard from them, that we kind of collectively called down and condensed and we'll put those against the tasks. 0:08:43

Interviewee: So when they need to schedule a user for an event, these are all the things they said about it, or these are the things we saw and the things that we felt. So then what's cool about that is we do that for every target user and you can map them out. So you say 'here are three target users, here are their days and here are all the intersections of their days'. So you can tell visually, "oh, you know what, when this person does something, you notice the next thing, these two people are affected." And then you can start isolating pain points and inefficiencies and you get these really nice overlay of what the system, not the digital system but just the users in the workplace and what their days look like. So when I'm mapping out the process, I'm thinking of conversations, that's a little talk bubble. 0:09:43

Todd: Ok, I like it. 0:09:44

Interviewee: And then I'm having doing some synthesis. Someone do a little post-it map. 0:09:51

Todd: Ooh, I see the post-its. 0:09:53

Interviewee: There you go. And then from that, we say, "ok, based on our research, this is what we say a persona or what this user looks like." We think of "what do they need? What are the tasks in their day and what do we need." So we pull out insights from that. This user really has trouble communicating with the other people on this team because they don't have the right communication tools setup or whatever that is. They need a better communication system. Once we have these needs, and bits and insights of who they are and what they need, then we can say, "all right, how can our products solve these needs?" Then we say, "we have some product ideas based on what their needs are, let's validate them." 0:10:55

Interviewee: Then we'll go back and talk to the users. We'll drop some wireframes and say, "all right, based on what you guys had said, we feel that here's a quick prototype, clickable prototype" And we'll use invision. We'll say, "Why don't you click around? What do you think about these things?" we'll do some user testing there. And then from that information, we'll further validate or dis-validate our product ideas and we'll do another product evolution but kind of the output of this 4 week on average DNF cycle as that you'll have Wireframes and then you'll have some personas. You'll know who your end user is. You'll have empathy drawn for your end user which is the whole goal. You'll have a problem that's been framed and validated. 0:11:38

Interviewee: So that way, it really de-risks development cause it's pretty great to come in to development and we know that when we're making product decisions, we can go back to this research. We're like, "oh, if we're going to do this or this, then what do they actually need? What was that they talked about that really indicated this is the right approach?" It also helps us speak a language that our users speak. Which is really important for the development team but all the other stakeholders involved in the process. 0:12:05

Interviewee: Words are everything, right? So we want people to be kind of on the same communication levels of talking how their users would talk, so that way it helps us draw all that empathy throughout the entire development process because you still need to draw on that, you know 6 weeks, 3 months, however long into the development cycle until you're releasing. Right to be able to have that insight of what they want. I think the cycle is... Let's just say that you've done some analysis and you've done some wireframing, and then after that, we'll talk to users again. After that, we'll do another set of wireframes. We'll also do some persona mapping here as well as making these wireframes. And then after we do that, we will create a feature list. 0:13:13

Todd: Yes. 0:13:14

Interviewee: We'll say, we're not gonna. . . What could the next you know 2 to 3 kind of epic areas. We'll say, let's give some insights here. Maybe, we have 3 insights and 3 needs. Let's do 3 feature ideas and how do these feature ideas map these needs? So we write these feature ideas and we'll say, down in the documents and user needs this and we'll help them achieve their goals. Business needs this and this will help them achieve their goals. We're always aligning user business needs throughout the entire process. So then when we get into development, we'll just make a little terminal, I wish I had multiple colors. 0:14:00

Todd: Next time, you'll have to bring your can. 0:14:03

Interviewee: So, when you're getting into the computer, when you're getting into development, you're able to have a backlog that's been built. You have the first couple of weeks of features. 0:14:17

Todd: Yes. 0:14:18

Interviewee: You have some ideas and you start building and then once you release something, which could be in a week or could be a couple of weeks depending on what you're doing. But once you get into the first bit of business value working software, then you can go and you could talk to your users again and then you learn from them and then you continue on. So at that point, what you're doing is you're building something and then you're measuring it. 0:14:45

Todd: Yes. 0:14:46

Interviewee: And then you're learning from it right? 0:14:48

Todd: Uh-huh. Yes. 0:14:50

Interviewee: And then you're going back to building, right? So that's what we're doing for this whole process. So the feedback loop is really important. When I'm thinking about the extent, the depth of all this research, then you don't need to have a DNF to do any of these research. You don't have to sell that in, you can do... So the training that our designers and our PMs have, and now we're trying to have our developers be exposed to it, as well. To say, "ok, developer, you get a client project. Development starts tomorrow, you can still talk to some users. You can still setup user testing." And you can do that throughout the entire process. So it's almost like this whole upfront part, with the DNF. You're making little DNFs through it, right? So whoops, it's really you're taking this and you're kind of doing a DNF cycle. 0:15:42

Todd: Yeah. 0:15:43

Interviewee: Whoops, and you're doing that here, and then you're going to do it again. So each week, you might be bi-weekly. And that is ideal, right? But sometimes you don't get the users? So there's a lot of, kind ofâĂę You have to be pretty scrappy with how you do this sometimes, you don't just get this nice set of users at your disposal, you know; but it's so important and I think that's something we do a good job with is convincing our clients, we wanna de-risk this for you so this is how we can do it. I mean go, it makes sense. It's very practical stuff. It's not rocket science honestly. 0:16:20

B.4 Interview 15 Abridged Transcript with Interaction Designer

Todd: My first question for you is very open ended. 00:02

Interviewee: Okay. 00:03

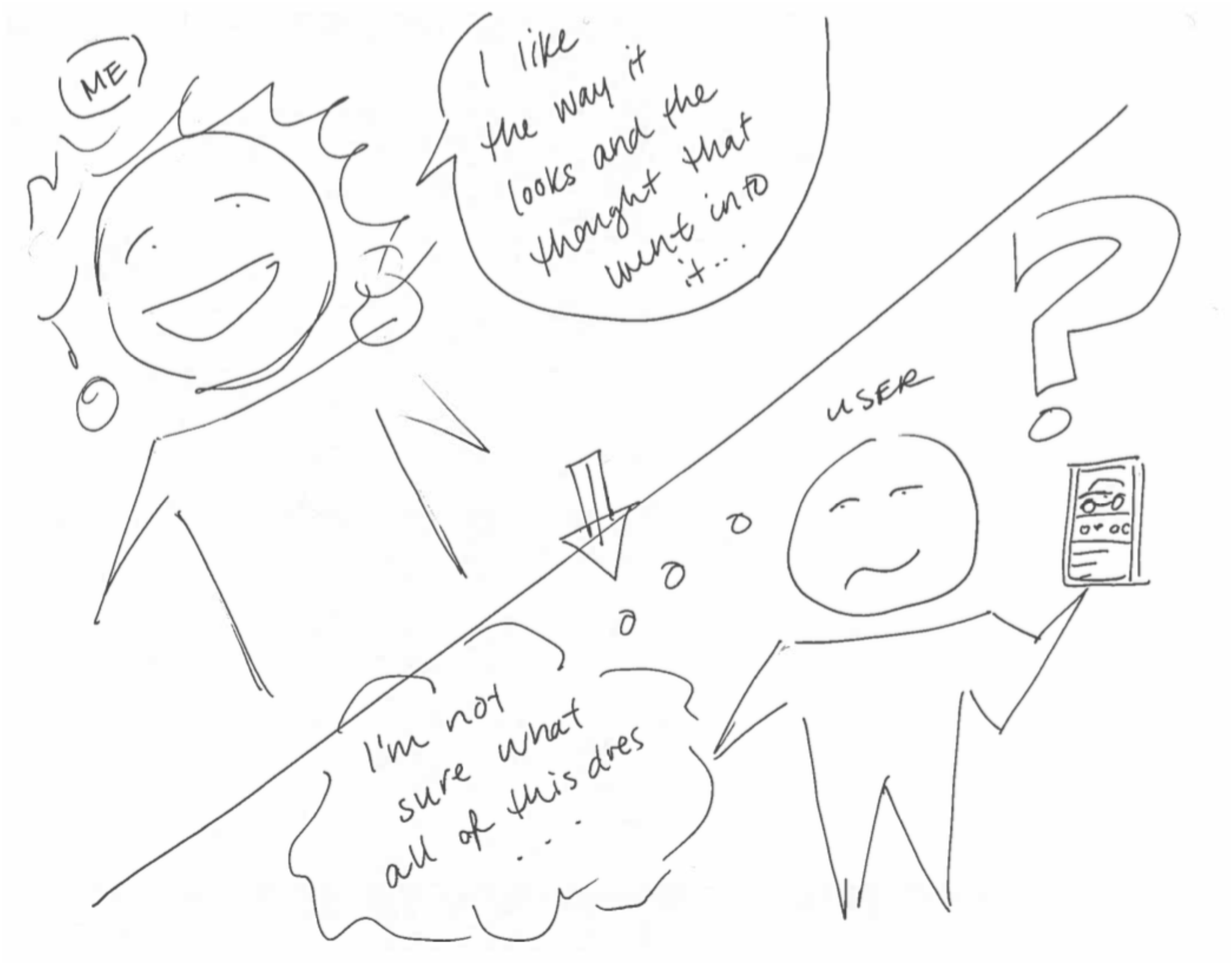

ME
I like the way it looks and the thought that went into it...
USER
I'm not sure what all of this does ...

Figure B.4: "2016-01-08 Interaction Designer's drawing of software development process"

Todd: There is no wrong answer. I was hoping if you could draw how you feel about the product. 00:09

Interviewee: Okay. This may get elaborate. 00:27

Todd: Fantastic. 00:29

Interviewee: This is my new pen so. [Pause] Here we are. I did sort of a story board-ish type of thing. 02:54

Todd: I love it. 02:55

Interviewee: So, do you want me to explain it? 02:58

Todd: Please. 02:58

Interviewee: Okay. On one hand superficially, I'm happy because aesthetically, I think it's nice. I think it's, when you compare it to some of the projects we do, it's been going on for so long and there's such a huge team of smart people. I feel like we got so much done and it's complex and interesting and there's lot of thought that went into it and it's a really robust app but I'm worried that even though it's pretty and we built a lot of features and the technology is cool, not all of it is necessarily useful for end users and I didn't even give the user name anything because I don't even know that we're designing for the right person all the time with some of our features. I don't know. So, I'm worried there were will be more confusion in the marketplace than I would like there to be in a product that I've worked on. 03:57

Todd: Anything else? 04:01

Interviewee: In the drawing or in general? 04:05

Todd: Both. 04:06

Interviewee: Not so much in the drawing. I guess the big question mark is just that I don't know, I think it's pretty usable but I'm worried there's going to be features for the users or going to be like what is this or why do I care or they'll have question marks around there's something really obvious to me like . . . I would like you heard a lot of feedback, I want a light telling me if my oil is low and that's just not something we could do because of constraints on the technology I think. So, I'm worried if people will look at it and say why is there all this stuff that I don't want and there's some stuff that maybe feels really obvious to some users that we haven't provided for one reason or another but overall, I still feel happy. I think we created a solid product. 04:59

Todd: Good. When you were describing your "you" picture, you used the word "superficially". I don't remember the exact word but something like given this superficially I feel happy about it and I was curious if there was like an under feeling of the product. 05:17

Interviewee: Yeah, currently to some degree, this is the under feeling. The superficial part is a little bit as a designer is a normal person walking around, you feel like people look at it and like it's so beautiful and that might be the beginning and the end of what they think of the app. They might not use it. Maybe, it's not for them. I feel like I could put it in a portfolio or take some of those App Store screens and show it to people and maybe like oh my God, this is the nicest product, you must have done a great job or your team must have worked really hard but if we built something that's really beautiful but doesn't meet the needs of our users, it's kind of I'm still superficial. I guess part of me is still happy it's beautiful at least or that there's parts of it that are really pretty but at the end of the day as a designer, it's kind of a big fail to build something that's pretty but not the right thing. It should be a big fail for everybody but especially as the designer, that's what you want to avoid. 06:21

B.5 Draw your view of Pivotal's software development process

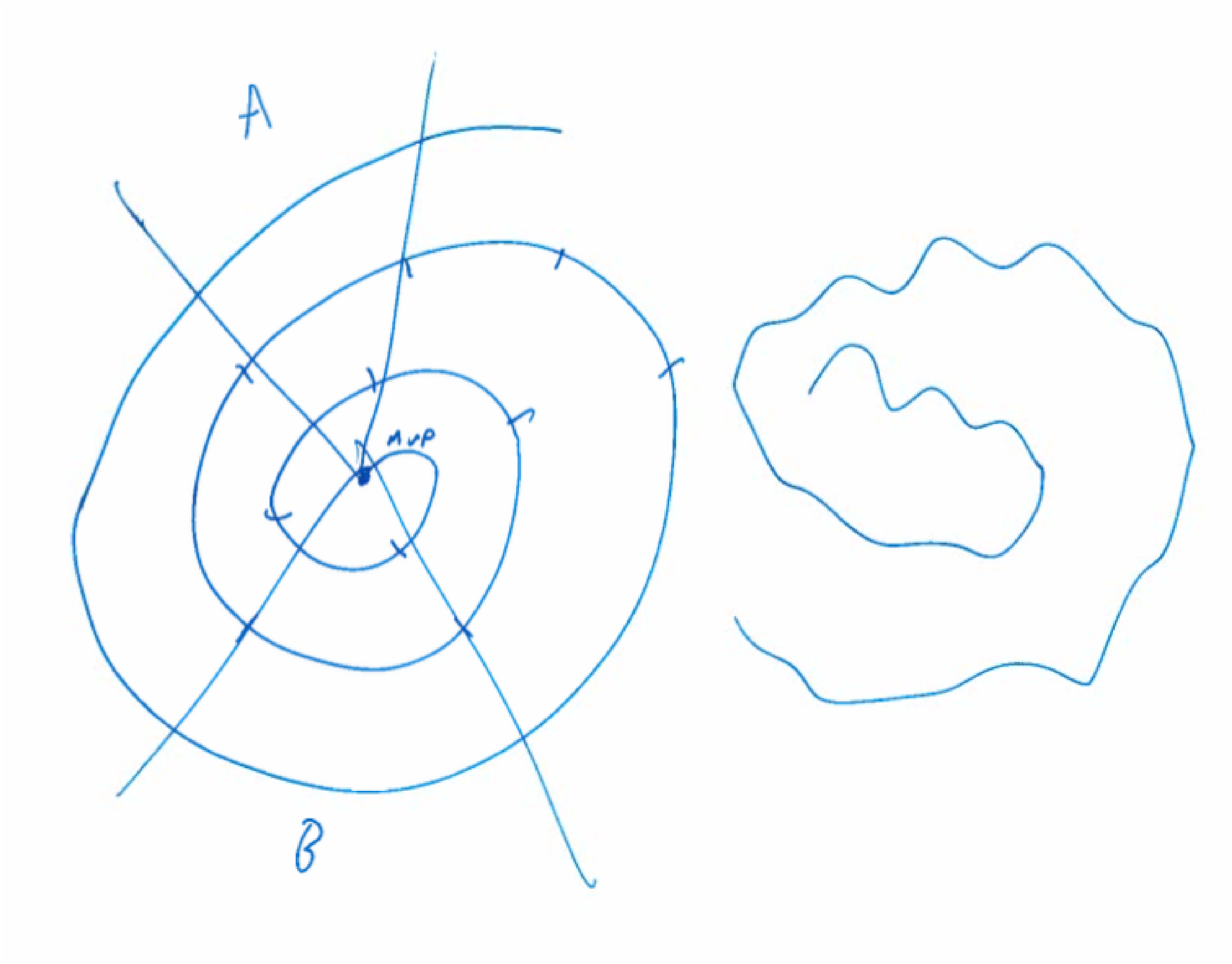


Figure B.5: "Interview 2: Product Manager's drawing of software development process"

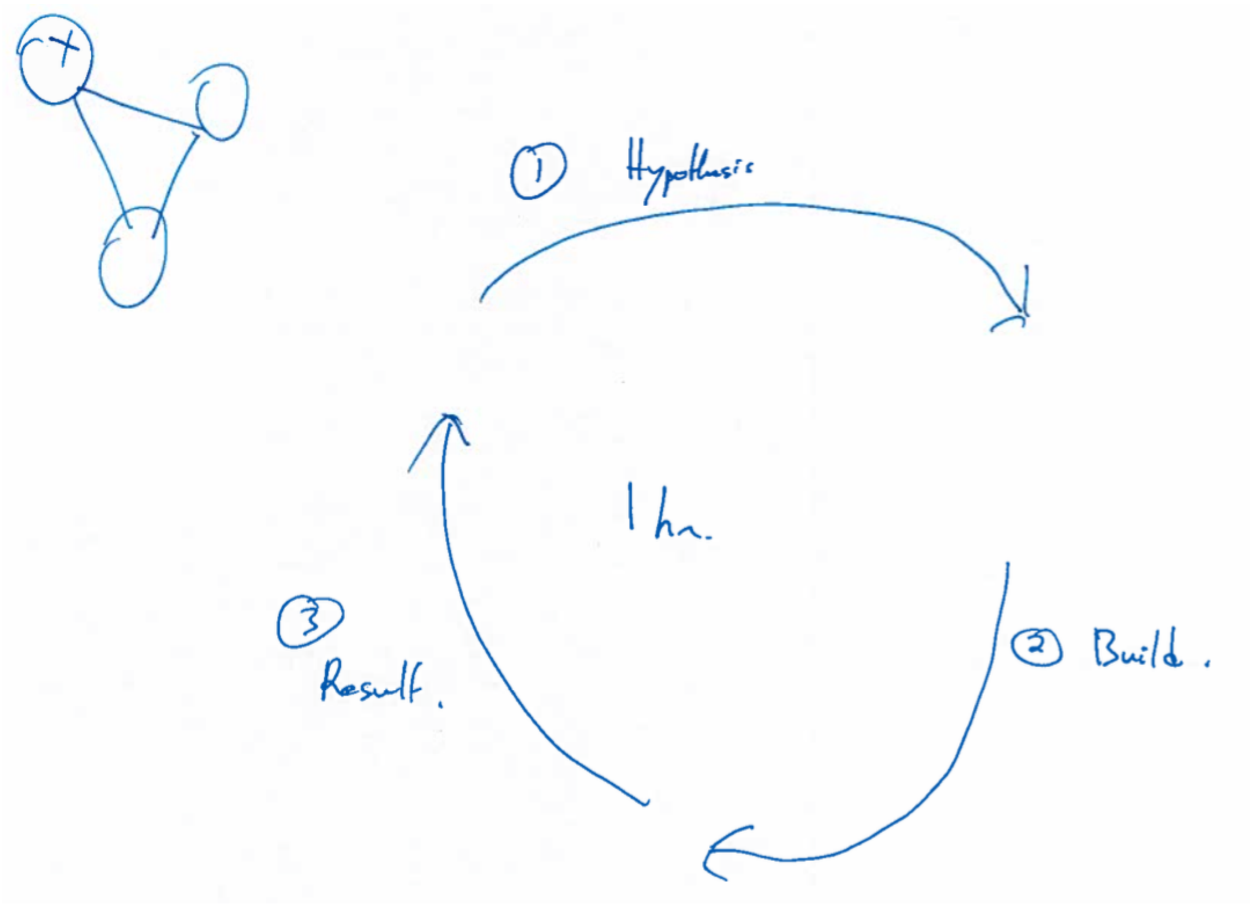


Figure B.6: "Interview 3: Product Manager's drawing of software project workflow"

Interviewee 3: I try to first come up with some type of hypothesis that I want to test. Then, I'm going to build something that's going to test this, then I'm going to try to get a result.

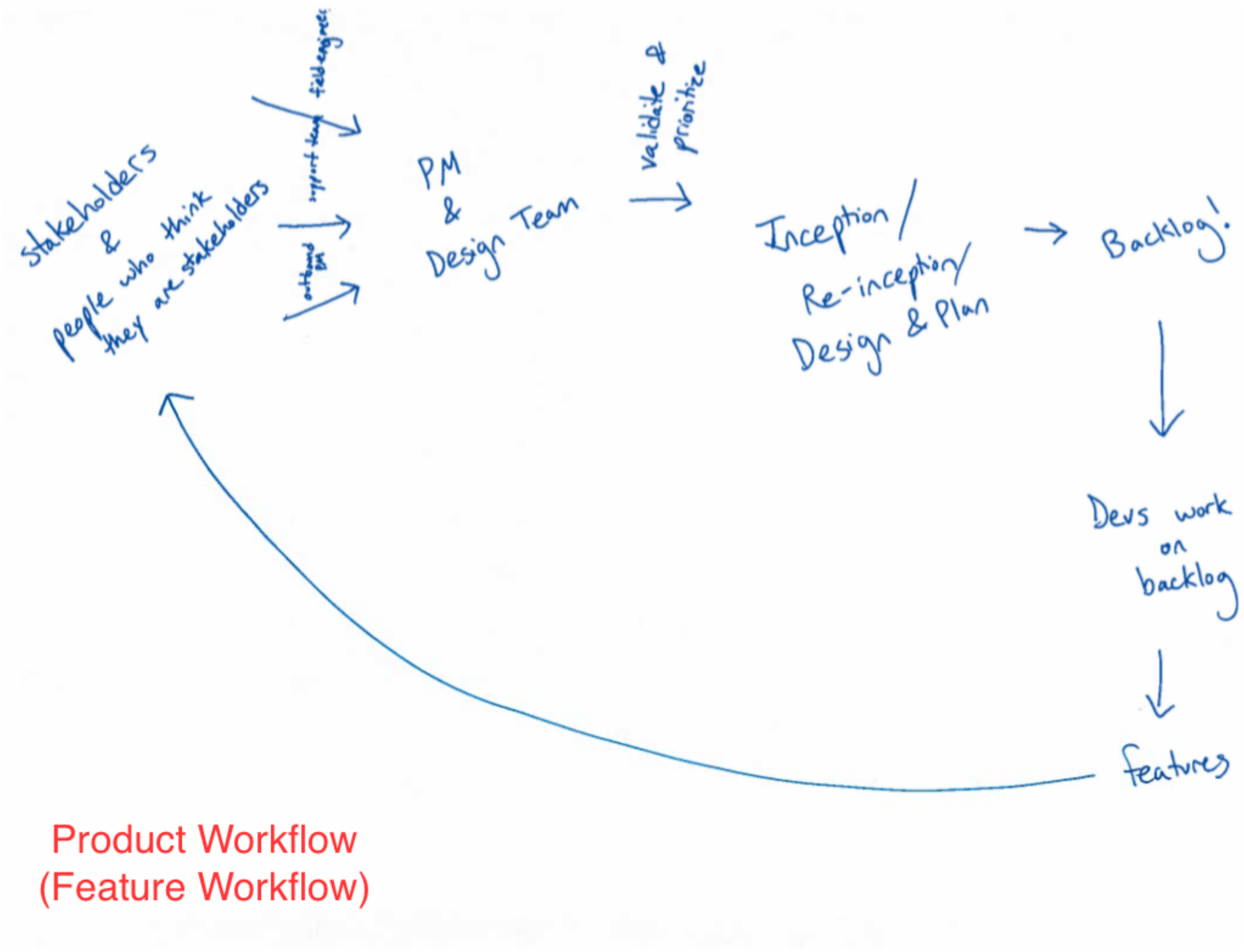

stakeholders & people who think they are stakeholders
outbound PM
support team
field engineers
PM & Design Team
validate & prioritize
Inception / Re-inception / Design & Plan
Backlog!
Devs work on backlog
features
Product Workflow (Feature Workflow)

Figure B.7: “Interview 5: Product Manager’s drawing of software development process”

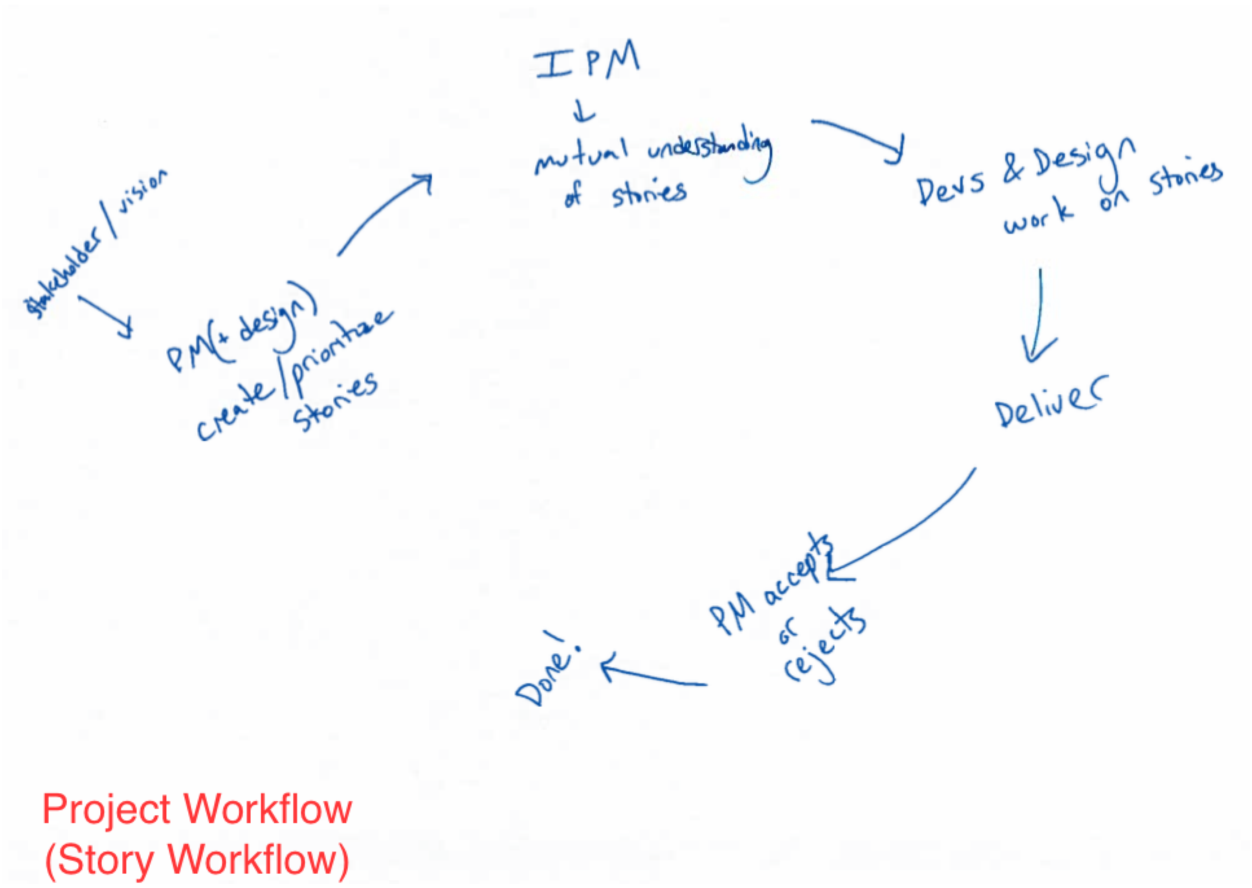

IPM
mutual understanding of stories
Devs & Design work on stories
Deliver
PM accepts or rejects
Done!
stakeholder / vision
PM (+ design) create / prioritize stories
Project Workflow
(Story Workflow)

Figure B.8: “Interview 5: Product Manager’s drawing of software development process”

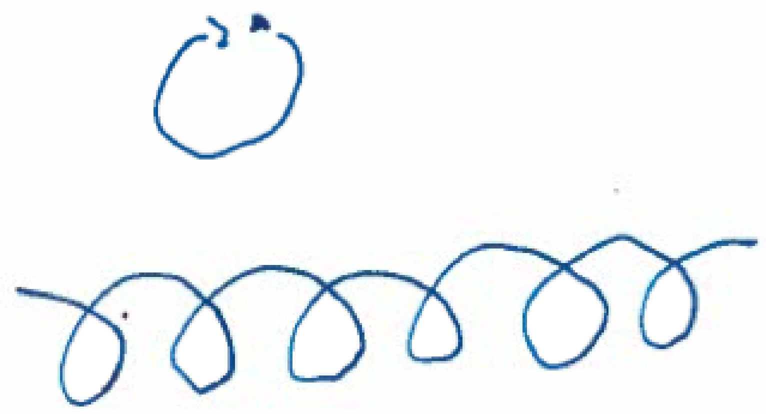


Figure B.9: "Interview 8: Software Engineer's drawing of software development process"

Interviewee 8: We iterate on stuff that we have a touch point and we're going to go away and do work and come back to that touch point. To me, it sort of looks more like a spring, you pull the slinky or you have a pig's tail that you stretched out. We are trying to do is orbit this idea of always working with each other. The red, green, refactor is a very tiny cycle that we do. We're trying to setup ourselves up, a check-in point where we can start somewhere and move a little bit and make sure we're okay and come back again and have another starting place and you can do that in 5 minutes with a test. Then you see us play that out at higher level of stories and then higher level again with our daily stand-ups, and higher level again with our retros and our check-ins there and higher level again with our inceptions. It's really about managing the feedback cycle or the check-in points to make sure that we're kind of all fluidly communicating about how we're affecting and doing things. Because the minute we stop communicating, is the minute we start to get off and be in the weeds somewhere.

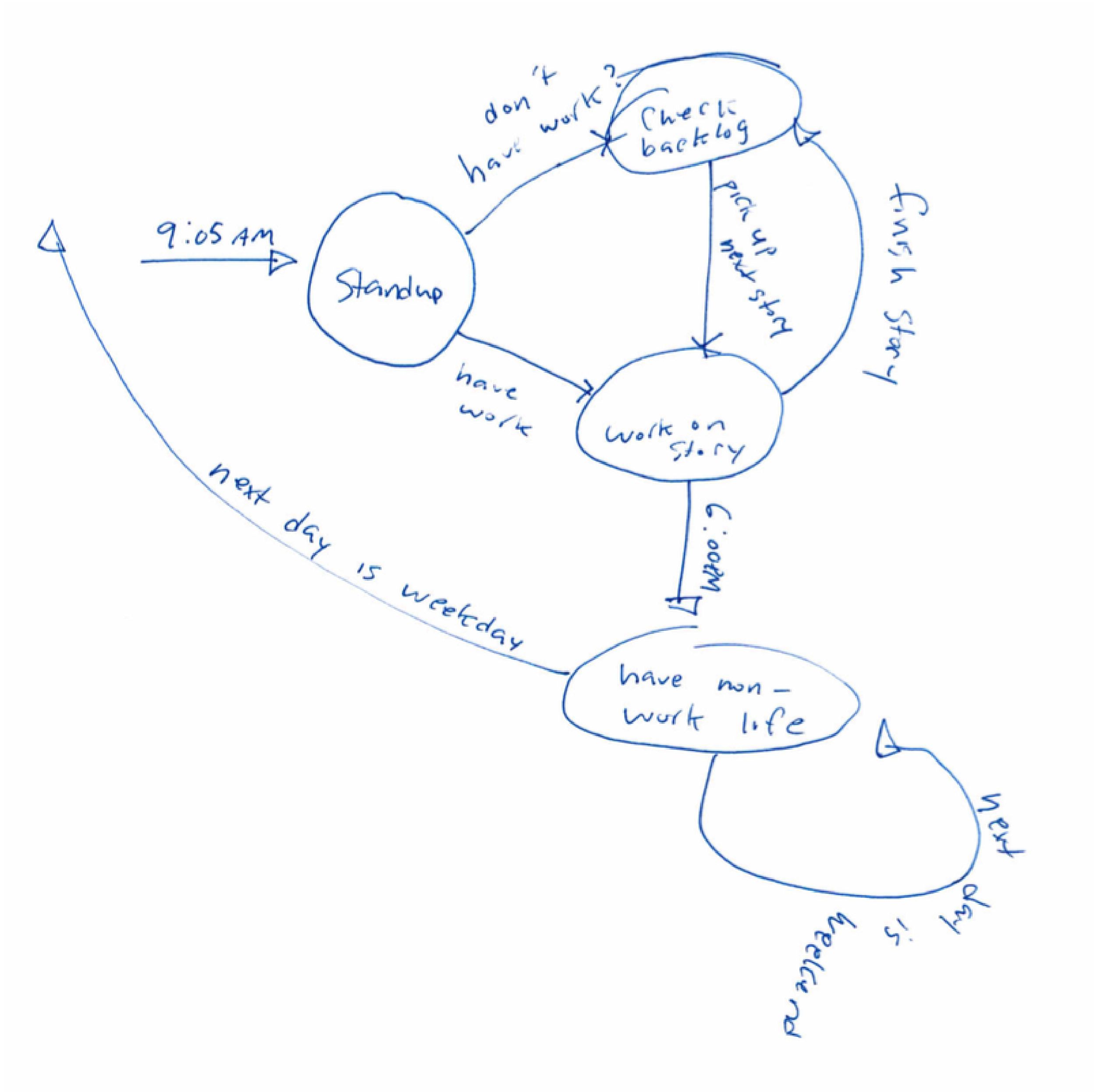


Figure B.10: “Interview 9: Software Engineer’s drawing of project workflow”

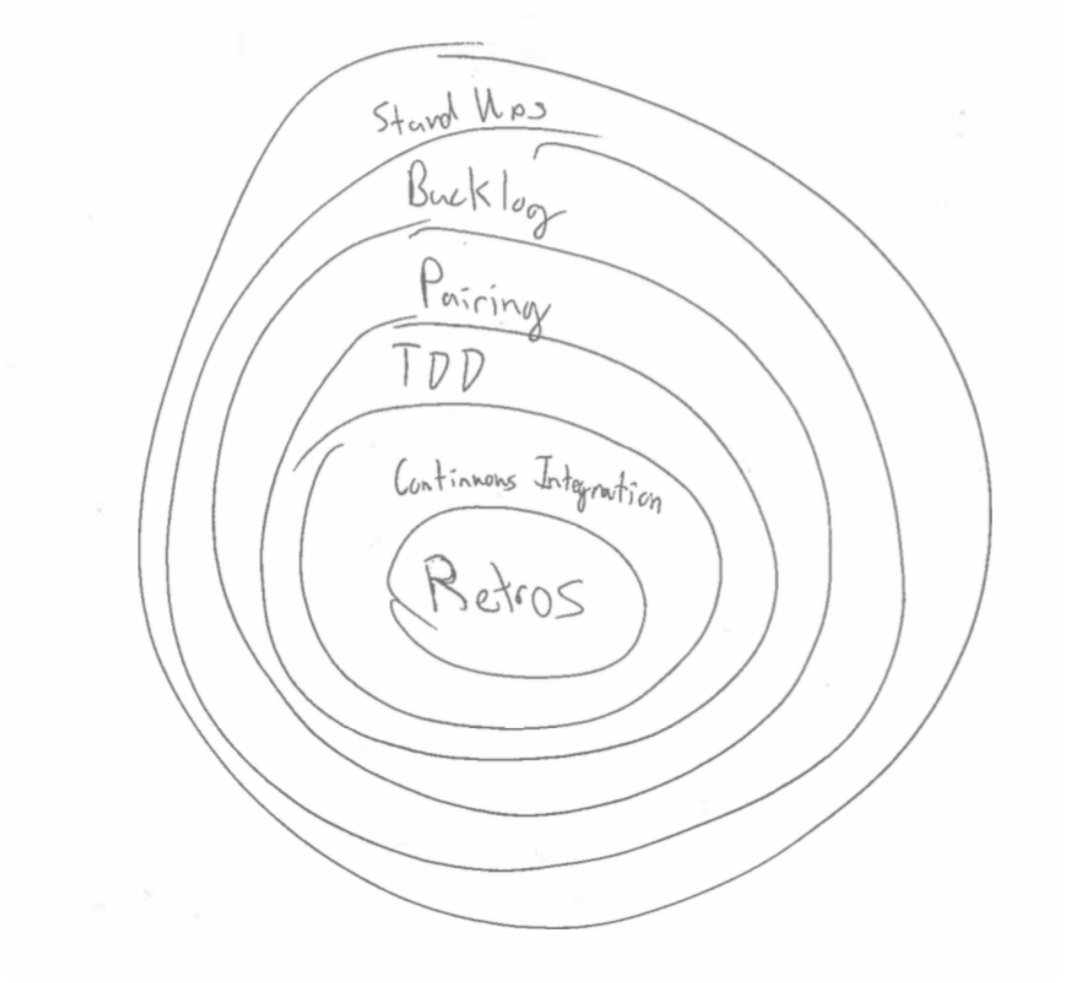


Figure B.11: "Interview 18: former software engineer's drawing for the Pivotal software development approach"

Interviewee 18: [The circles are arranged from an inner core practices to outer rings that are more negotiable.] These are the things I think are core to how Pivotal Labs does software development from the most important. Basically, if some client wanted to drop things outside of it, this will be the order that I think we'd agree to have it be dropped. I don't think we'd ever drop retros but I think we might drop the backlog or something that we probably still would.

B.6 Draw how you feel about your current product?

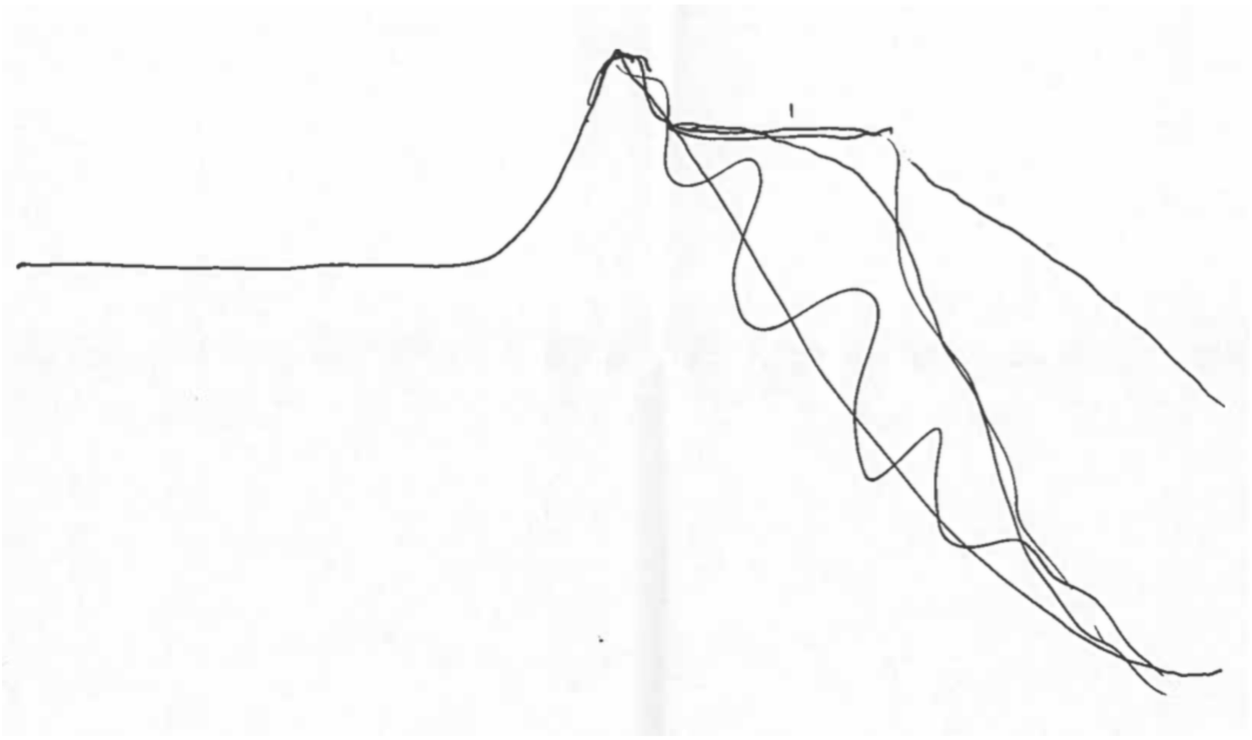


Figure B.12: "Interview 16: interaction designer's drawing for 'how you feel about your current product?"'

Interviewee 16: The peak was very like a successful moment for Pivotal. It's really cool to see how many people appreciated what we've done. Against all odds, we released this app. I felt proud to be on that team. . . .

[The graph goes down] as the design started getting cut. The only thing we're doing is bugs. We designed all these cool features that we're excited about. Almost none of them are getting into the app. It's been almost every week that a feature in the new design is no longer in [the release]. We don't need [what I worked on] anymore. It feels very frustrating as a designer. I can't affect that that much.

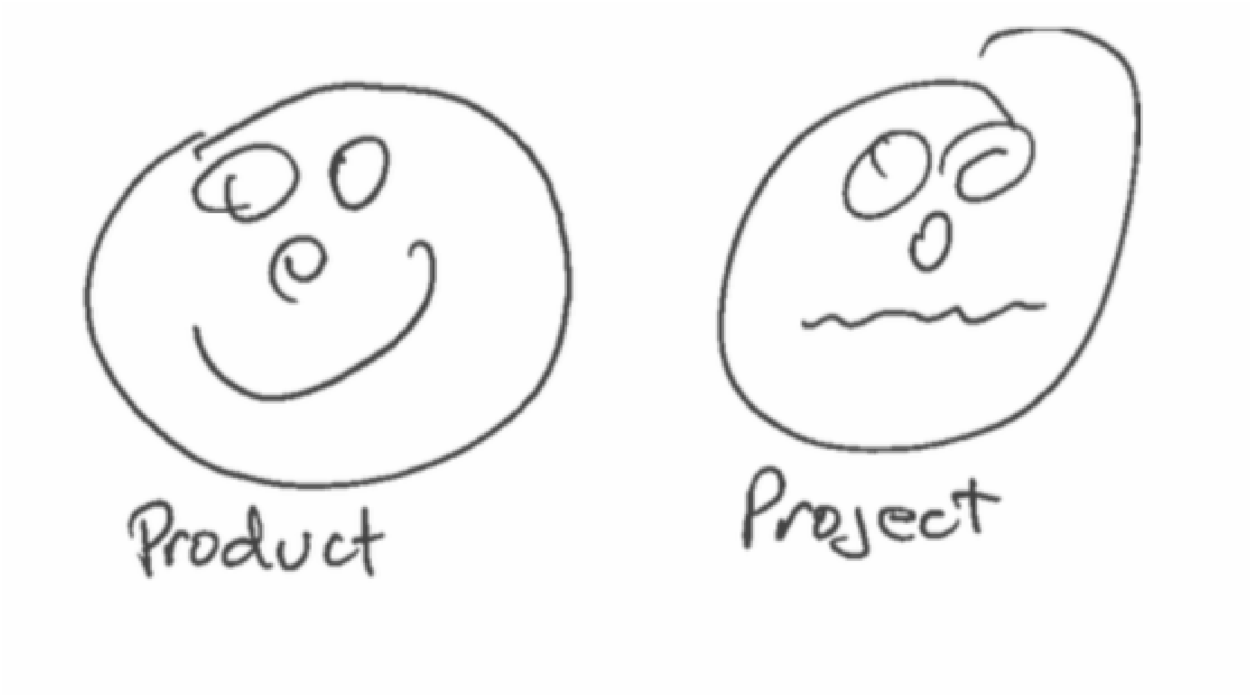


Figure B.13: "Interview 21: Product Manager's drawing for 'how you feel about your current product"'

B.7 Draw how you feel about the code

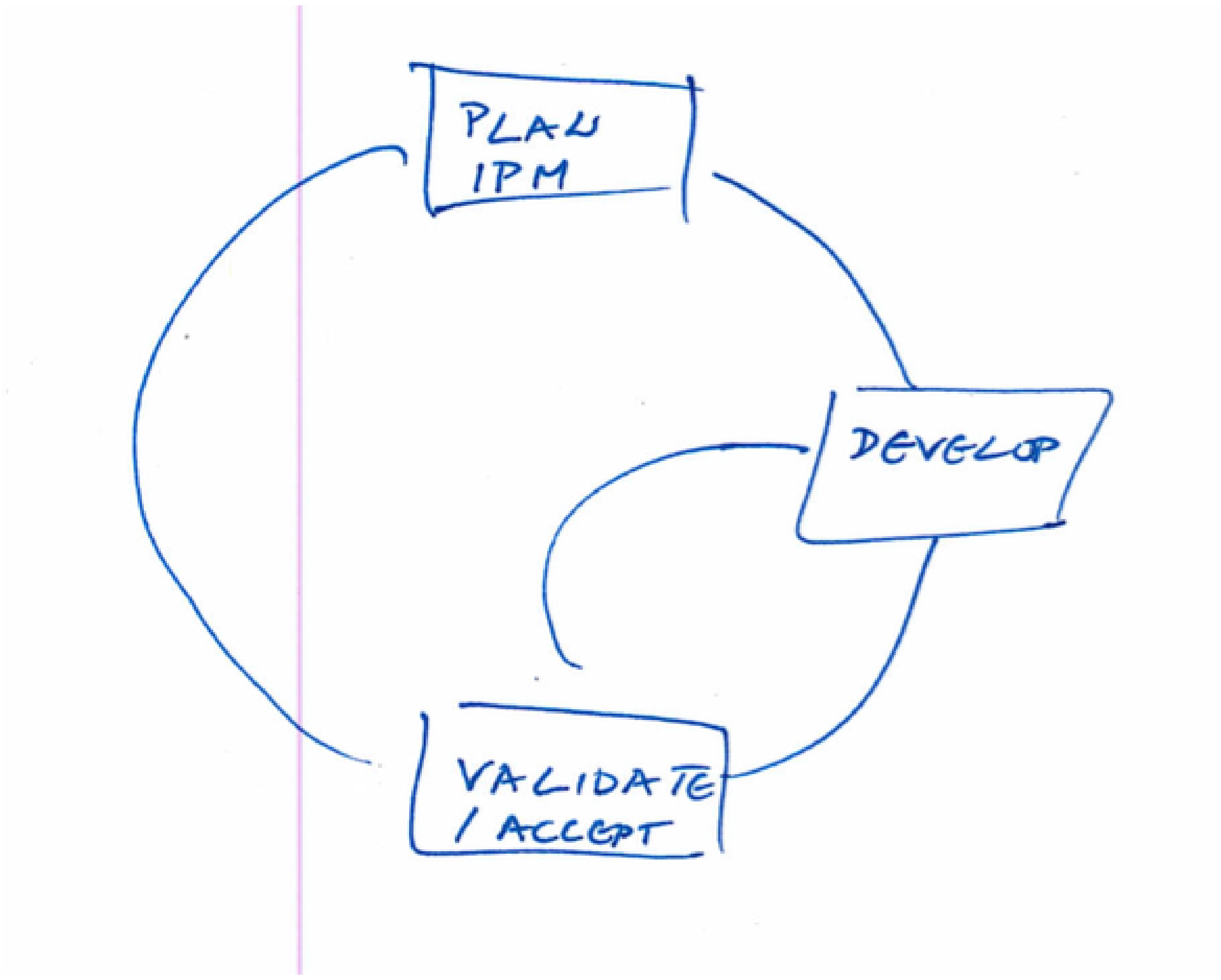


Figure B.14: "Interview 14: Software Engineer's drawing for 'how do you picture the code?"'

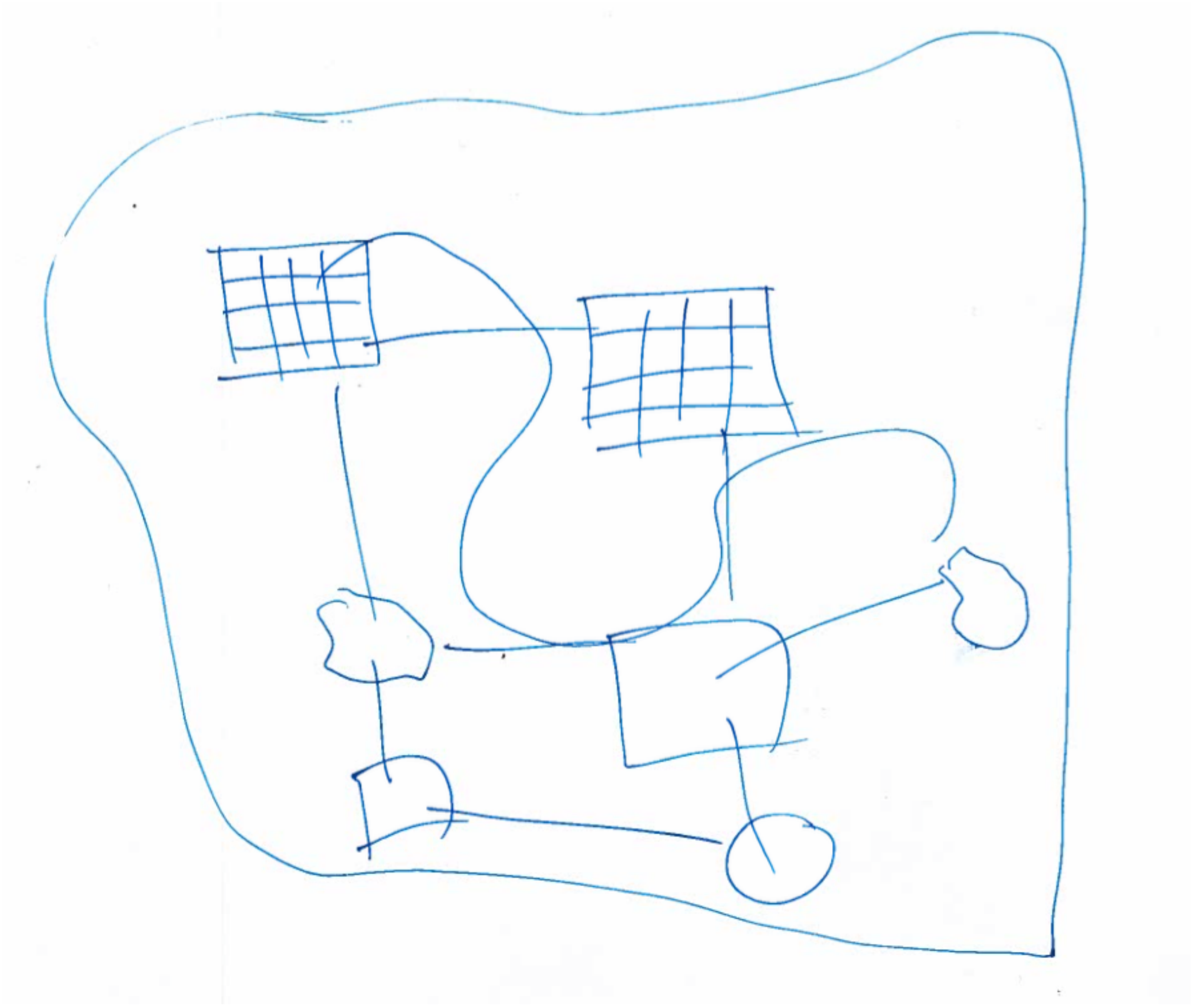


Figure B.15: "Interview 10 Software Engineer's drawing for 'how you think of the code on this project?"'

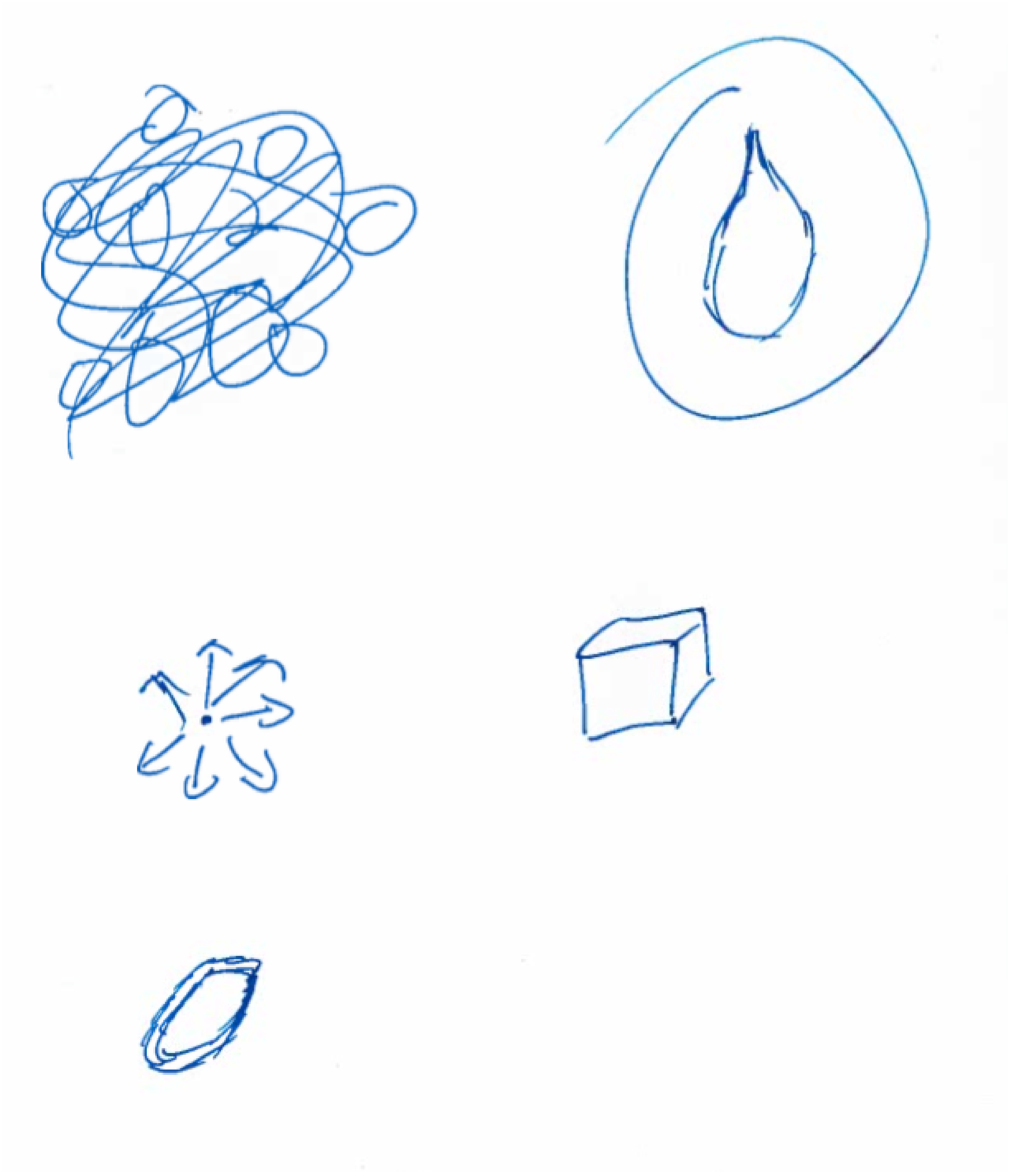


Figure B.16: "Interview 11: Software Engineer's drawing for 'how you think about the code on this project?"'

Interviewee 11: It's kind of complicated. We have so many different pieces and this is what the connection feels like to me in my head where everything is jumbled together but at the same time, I do feel like the structure is clean but I'm having a hard time thinking of what to draw for like what represents clean. . I'm just going to draw a water droplet to show that it looks clean.

It's more than just it's complicated, it's expanding a lot so in my head. I'm thinking more like a big bang kind of thing where it starts out very small and now it just keeps expanding and growing and then we're adding a bunch of features.

It's solid and clean. I feel like the project is very well tested so there's less chances of major breakings. It is solid. So I draw a rock cube.

Our codebase is pretty young, pretty flexible in ways that when we want to do refactors, it's not super complicated and not super hard to do. We are pretty good at like separating all the logic. It was really easy to do refactoring on the stuff that we want to do. So, I guess it's pretty flexible of something flexible. So I draws a rubber band.

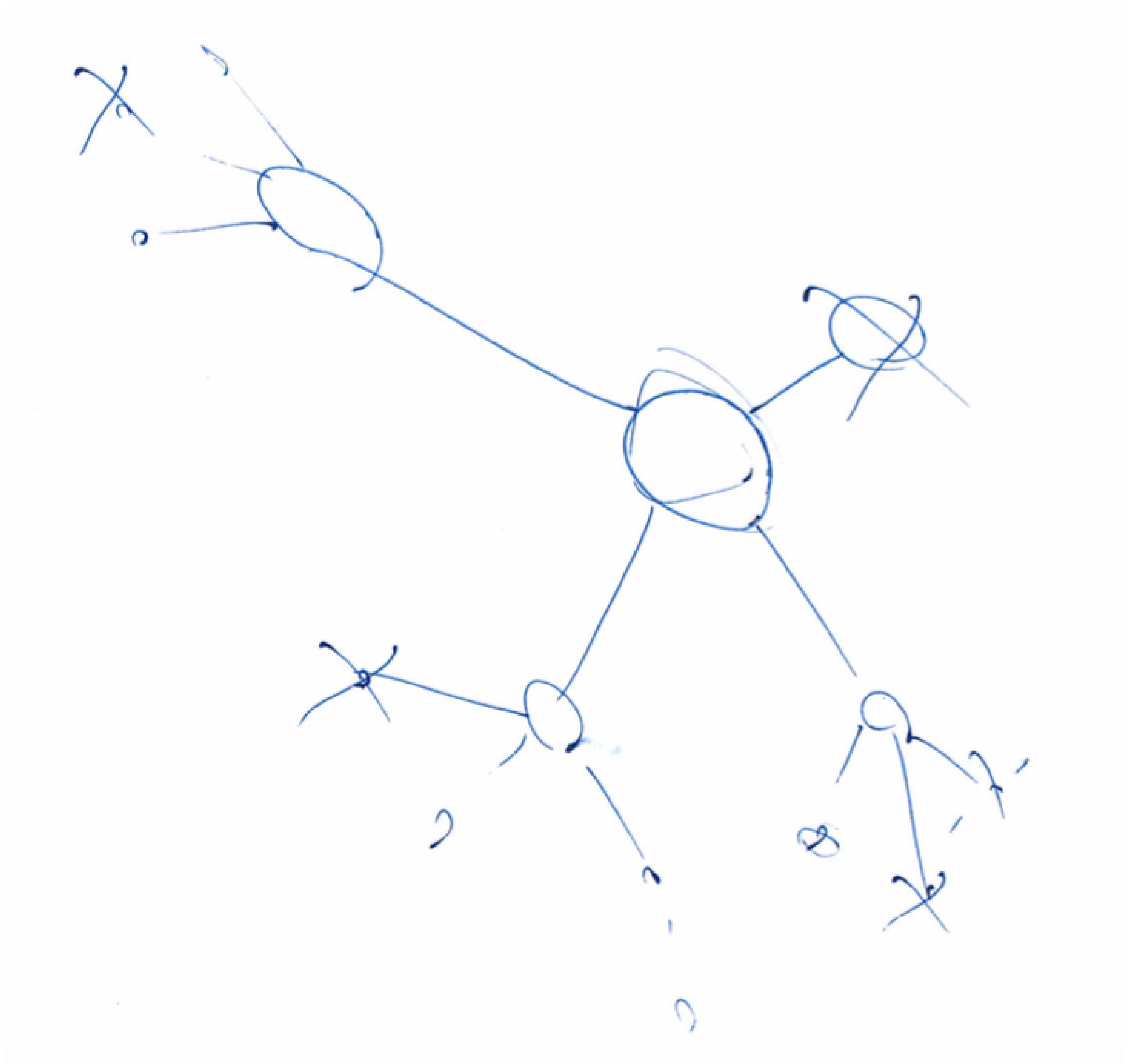


Figure B.17: "Interview 13: Software Engineer's drawing for 'how you think about the code on this project?"'

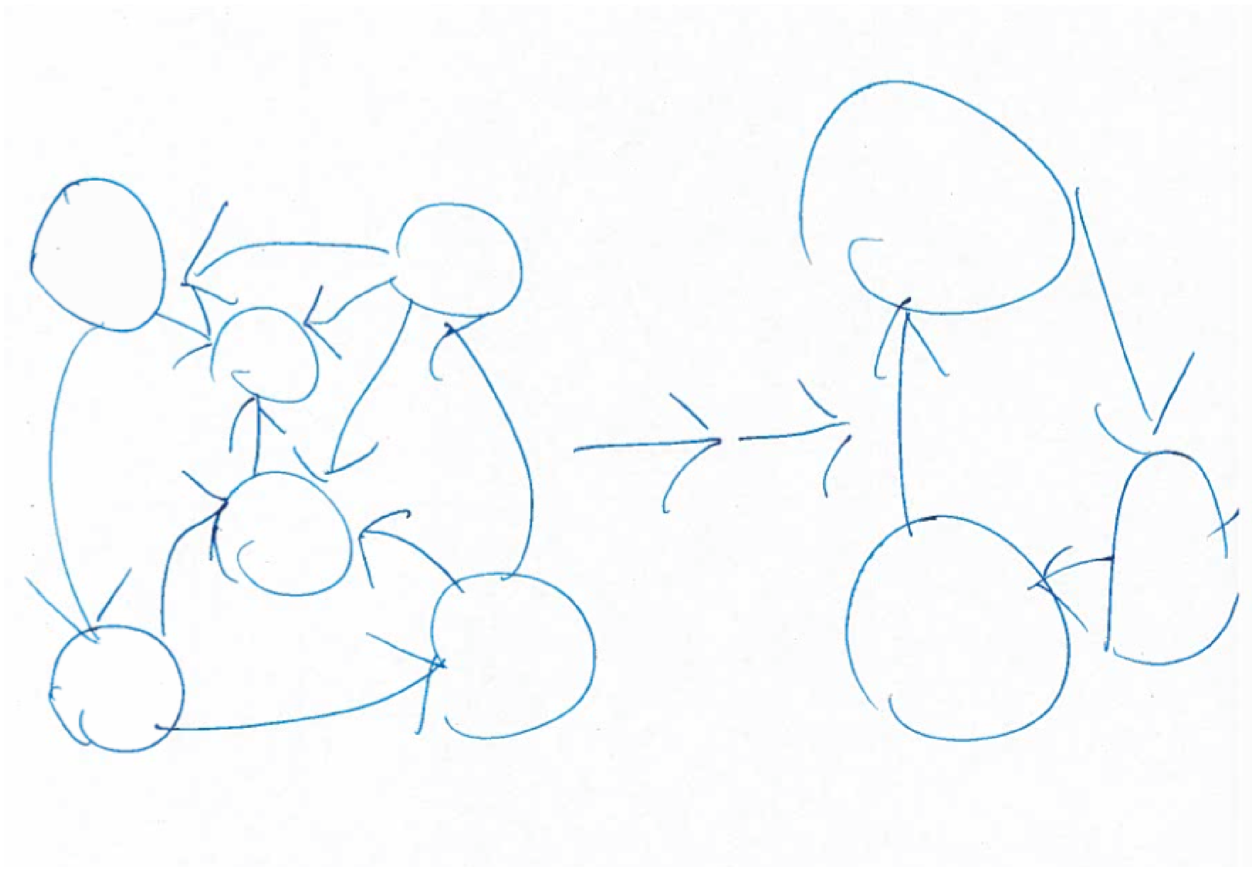


Figure B.18: "Interview 17: Software Engineer's drawing for 'How you feel or how you think about the code"'

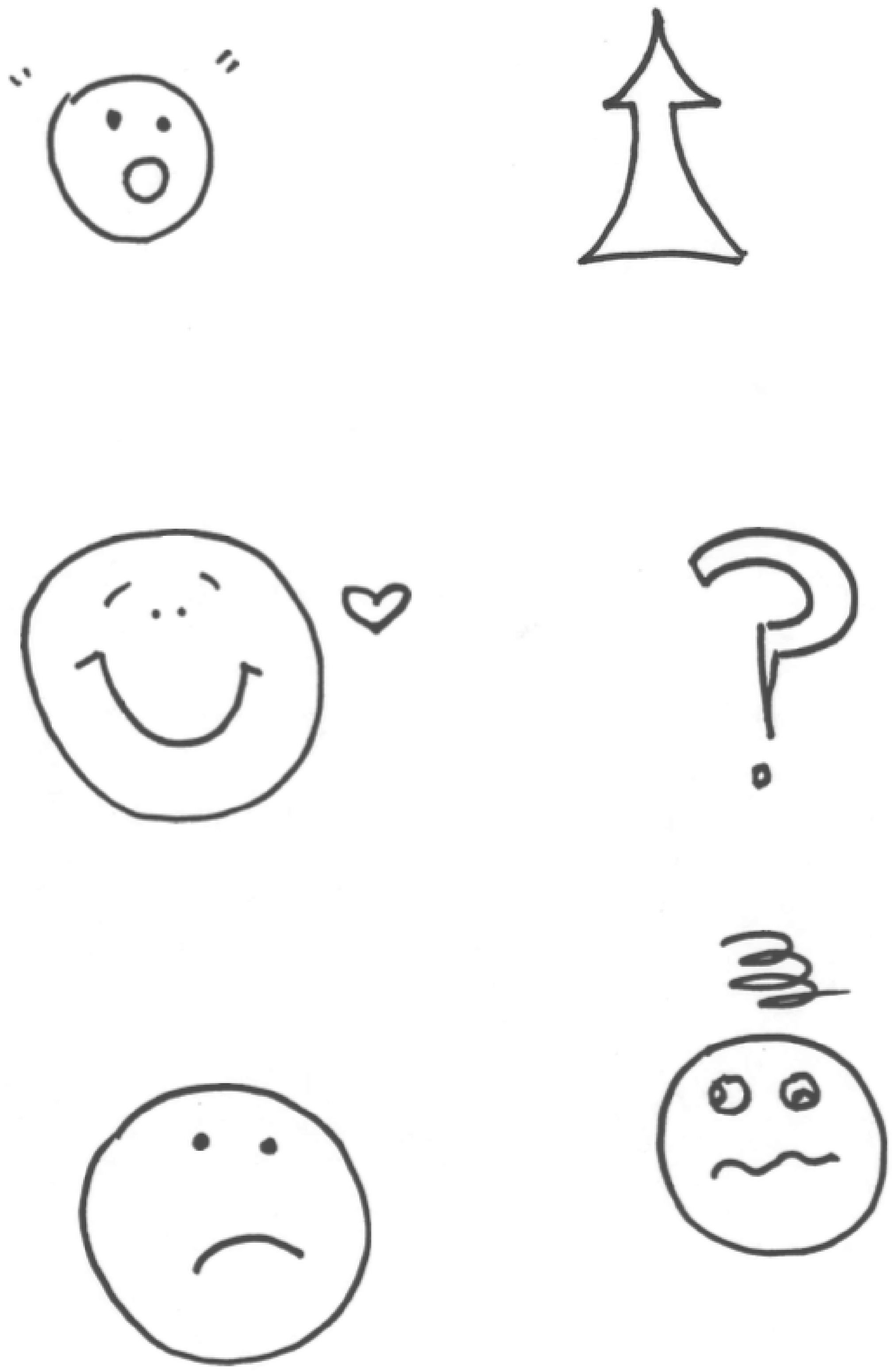


Figure B.19: "Interview 25: Interaction Designer's drawing for 'how you feel about your current product"'

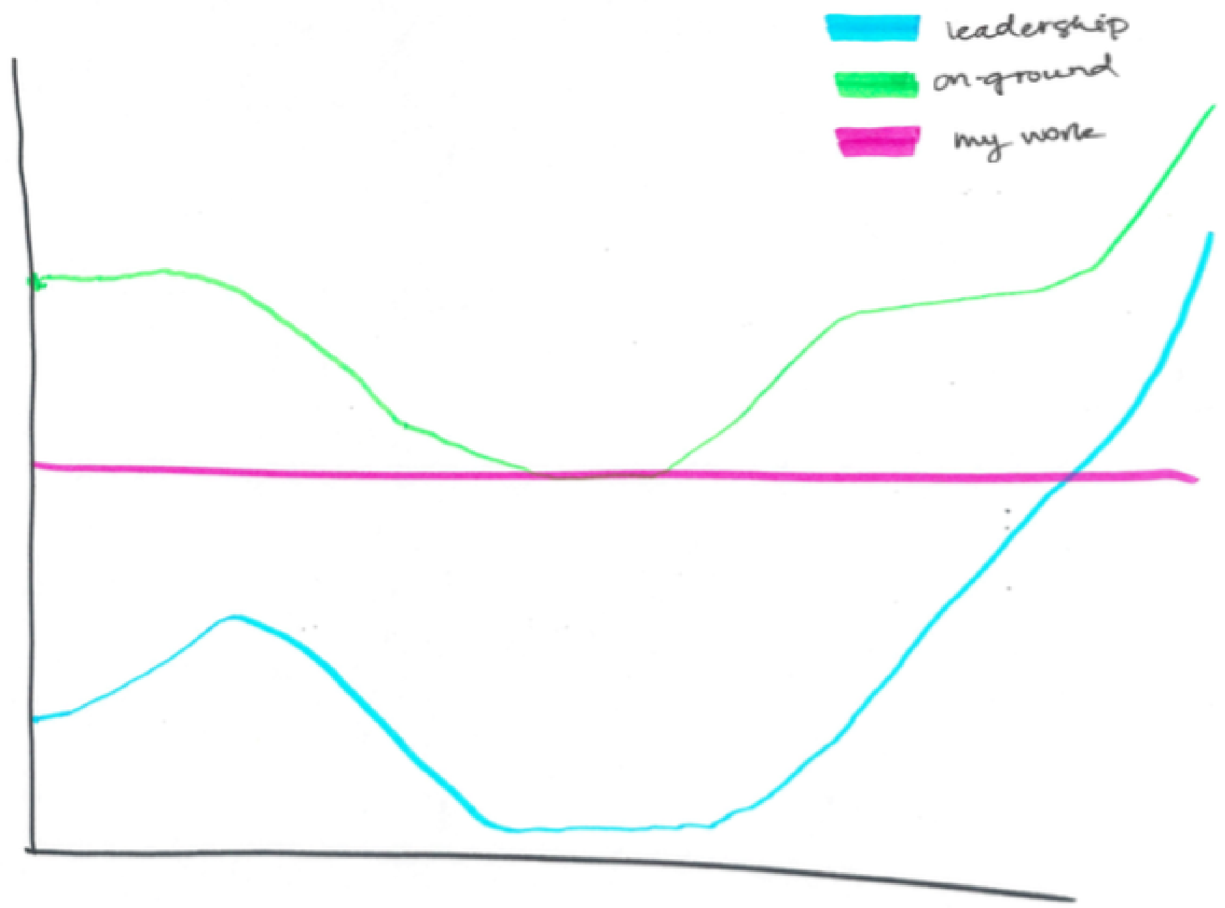


Figure B.20: "Interview 25: Interaction Designer's drawing for 'how you feel about your current product"'

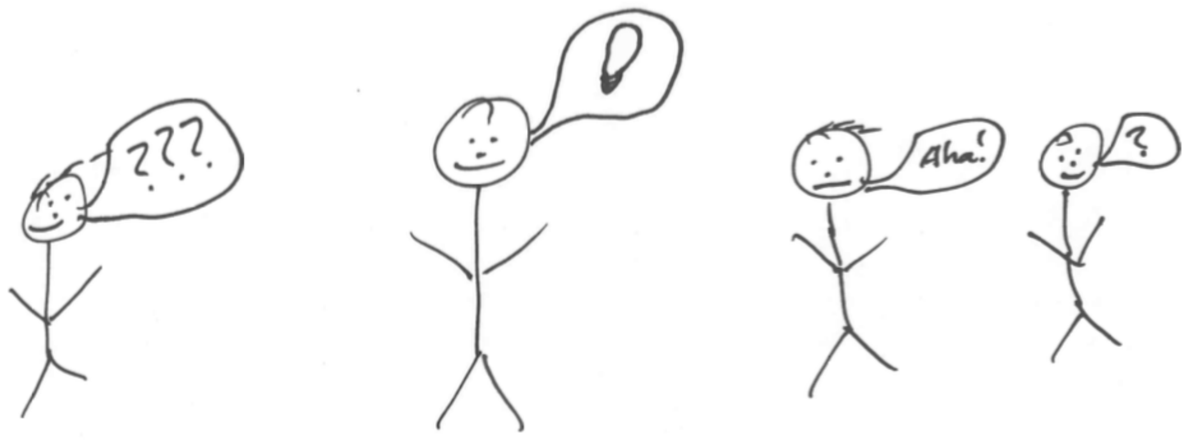


Figure B.21: "Interview 26: Interaction Designer's drawing for 'How you feel or how you think about the code"'

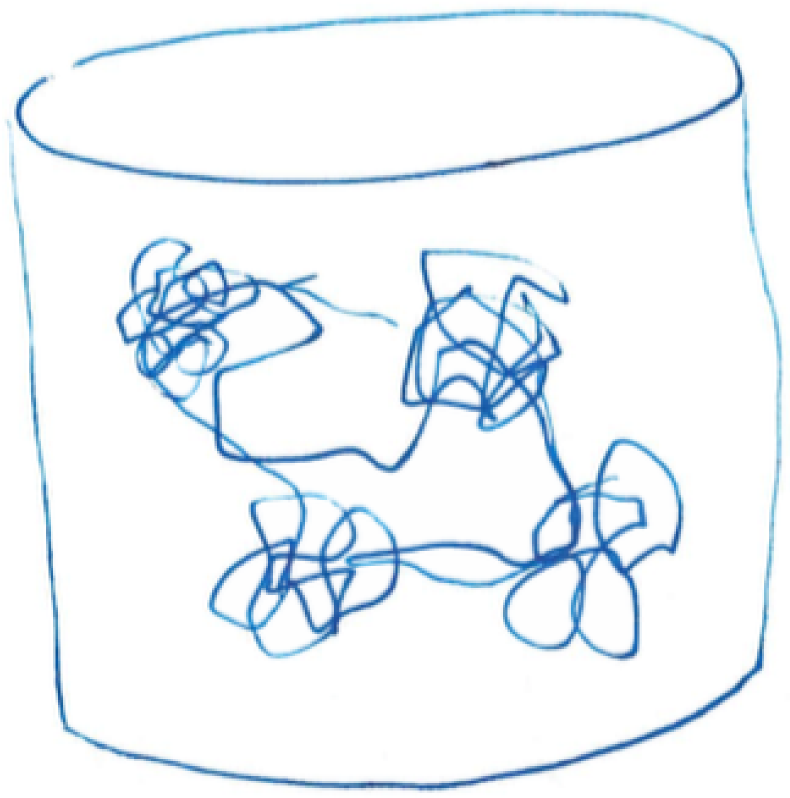


Figure B.22: “Interview 28: Software Engineer’s drawing for ‘How you feel or how you think about the code”’

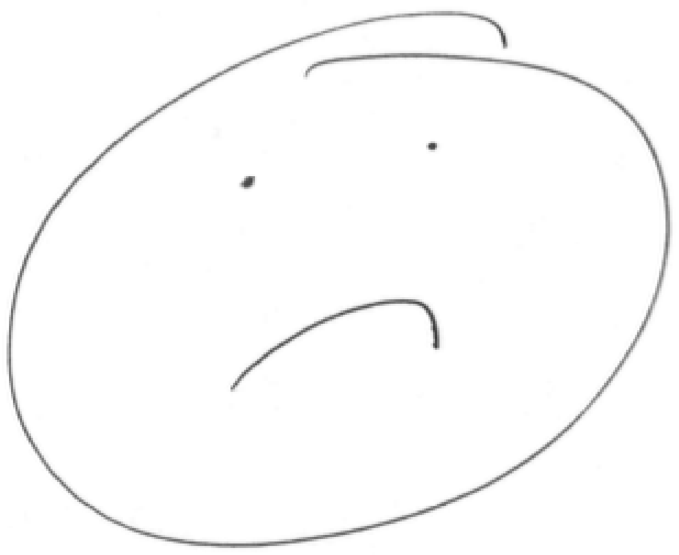


Figure B.23: "Interview 29: Software Engineer's drawing for 'How you feel or how you think about the code"'

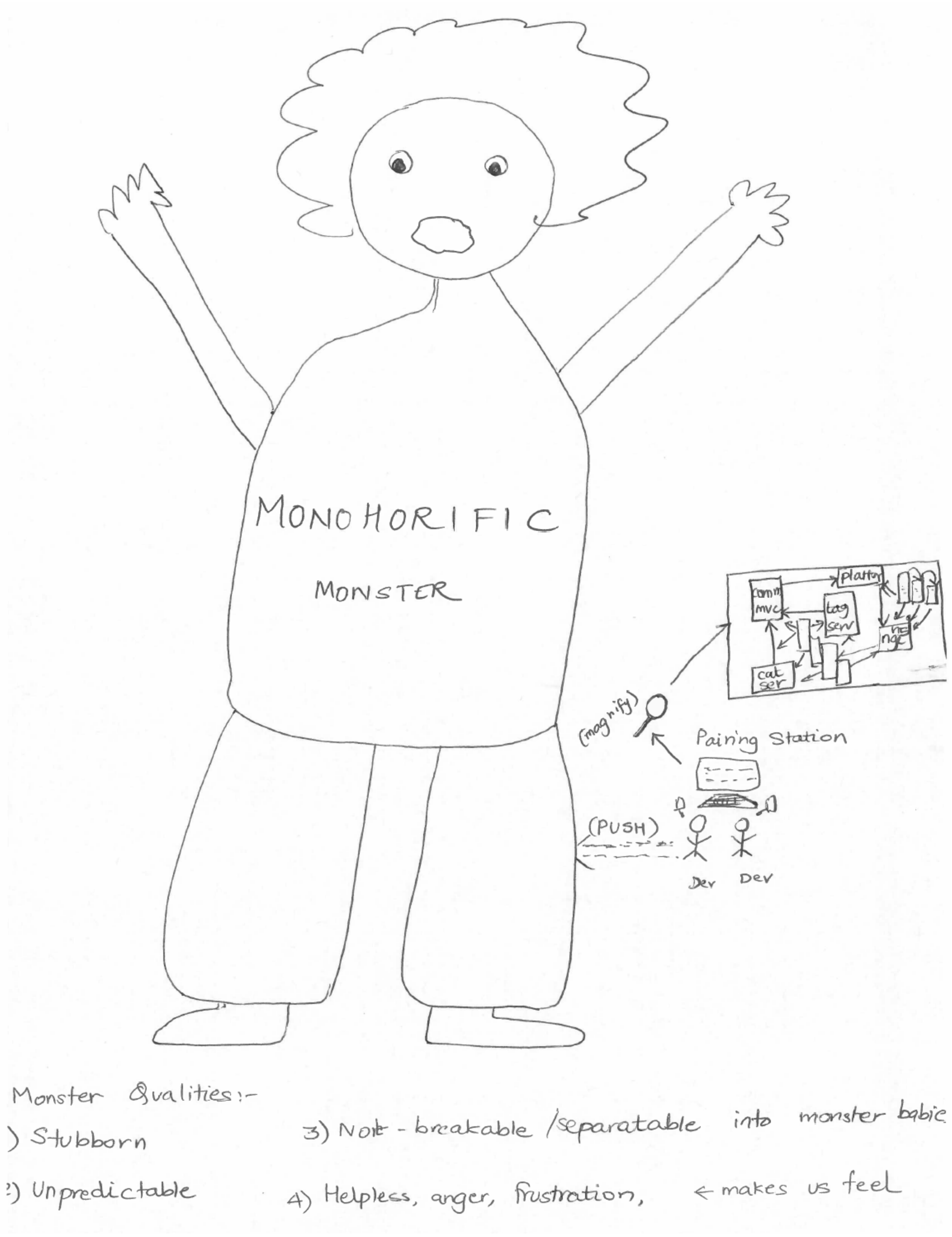


Figure B.24: "Interview 31: Software Engineer's drawing for 'How you feel or how you think about the code"'

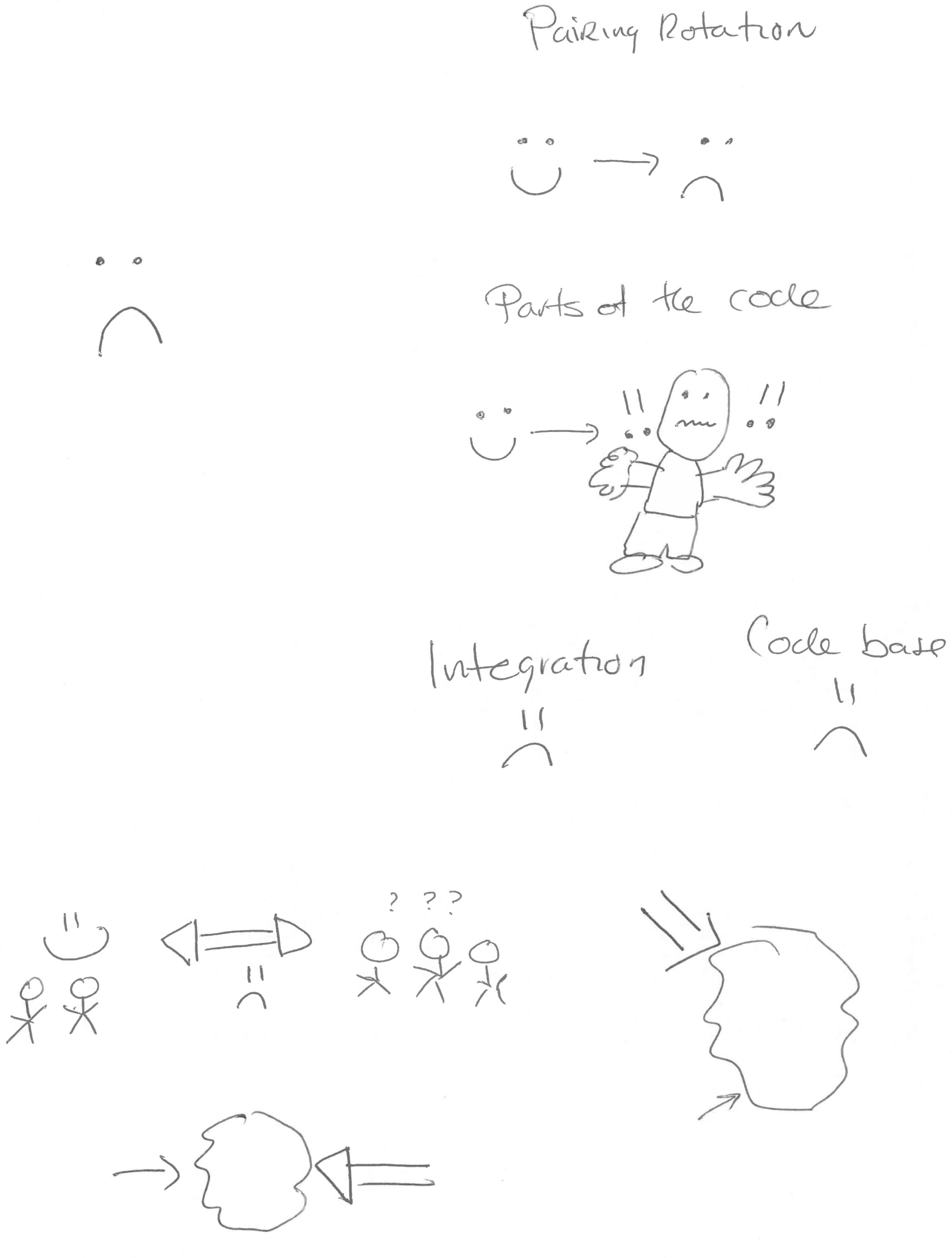


Figure B.25: "Interview 32: Software Engineer's drawing for 'How you feel or how you think about the code"'

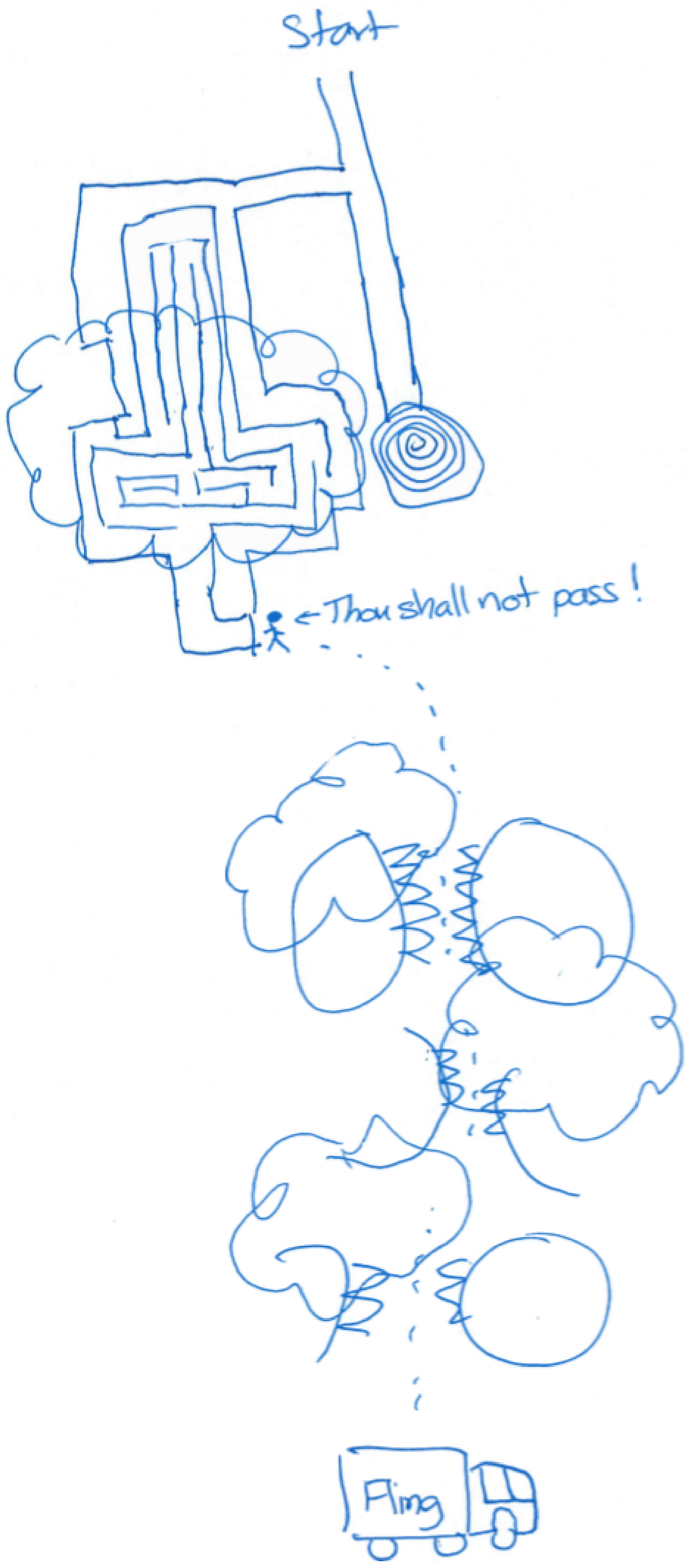


Figure B.26: "Interview 33: Software Engineer's drawing for 'How you feel or how you think about the code"'

Bibliography

[1] N. Alaverdyan. Pair programming matrix / board, 2010. URL: http://alaverdyan.com/readme/2010/12/pair-programming-matrix-board/. 48

[2] N. B. Ali, K. Petersen, and K. Schneider. Flow-assisted value stream mapping in the early phases of large-scale software development. *Journal of Systems and Software*, Jan. 2016. 81, 93

[3] D. J. Anderson. *Kanban: successful evolutionary change for your technology business*. Blue Hole Press, 2010. 78

[4] A. R. Artino Jr. Cognitive load theory and the role of learner experience: An abbreviated review for educational practitioners. *Aace Journal*, 16(4), 2008. 87

[5] K. Awati. Increasing your team's bus factor, 2008. URL: https://eight2late.wordpress.com/2008/09/03/increasing-your-teams-bus-factor/. 48

[6] D. J. Beal, R. R. Cohen, M. J. Burke, and C. L. McLendon. Cohesion and performance in groups: a meta-analytic clarification of construct relations. *Journal of Applied Psychology*, 88(6):989, 2003. 73

[7] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999. 6, 48, 60, 66

[8] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. viii, 4, 6, 7, 8, 9, 10, 11, 12, 14, 41, 47, 48, 60, 66

[9] 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. 57, 67

[10] K. A. Bollen and R. H. Hoyle. Perceived cohesion: A conceptual and empirical examination. *Social forces*, 69(2):479–504, 1990. 73

[11] K. Charmaz. *Constructing Grounded Theory*. SAGE Publications, 2014. 17, 18, 19, 20, 21, 23, 24, 25, 26, 27, 29, 30, 31, 33, 37, 38, 40, 42, 44, 75, 92

[12] 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. 59

[13] J. O. Coplien. A generative development process pattern language. In *Proceedings of Pattern languages of Program Design*, PLoP, 1994. 48

[14] R. Davies and L. Sedley. *Agile Coaching*. Pragmatic Bookshelf, 2009. 48

[15] E. Derby and D. Larsen. *Agile Retrospectives: Making Good Teams Great*. Pragmatic Bookshelf, 2006. 43

[16] R. Formanek. Why they collect: Collectors reveal their motivations. *Interpreting objects and collections*, 1994. 67

[17] M. Fowler. Code ownership, 2006. URL: `http://martinfowler.com/bliki/CodeOwnership.html`. 66

[18] S. Freeman and N. Pryce. *Growing object-oriented software, guided by tests*. Pearson Education, 2009. 60

[19] B. G. Glaser. *Theoretical Sensitivity: Advances in the Methodology of Grounded Theory*. Sociology Press, 1978. 17, 20, 21, 26, 34, 35, 38, 40, 44

[20] B. G. Glaser. *Basics of grounded theory analysis: emergence vs forcing*. Sociology Press, 1992. 19, 20, 28, 49

[21] B. G. Glaser. *Doing Grounded Theory: Issues and Discussions*. Sociology Press, 1998. 23, 32, 63, 75, 93

[22] B. G. Glaser. *The grounded theory perspective: Conceptualization contrasted with description*. Sociology Press, 2001. 21

[23] B. G. Glaser. All is data. *Grounded Theory Review*, 6(2), 2007. 20

[24] B. G. Glaser and A. L. Strauss. *Awareness of dying*. Transaction Publishers, 1966. 19

[25] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Sociology Press, 1968. 18, 19, 20

[26] J. Grudin and J. Pruitt. Personas, participatory design and product development: An infrastructure for engagement. In *Participatory Design Conference*, 2002. 82

[27] S. Isaacs. Social development in young children. *British Journal of Educational Psychology*, 1933. 67

[28] 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. 48

[29] W. James. *The Principles of Psychology*. Holt, 1980. 67

[30] 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. 48

[31] J. T. Jost, M. R. Banaji, and B. A. Nosek. A decade of system justification theory: Accumulated evidence of conscious and unconscious bolstering of the status quo. *Political Psychology*, 25(6), 2004. 78

[32] J. Karabanow. Getting off the street exploring the processes of young people's street exits. *American Behavioral Scientist*, 51(6):772–788, 2008. 30

[33] M. Khurum, K. Petersen, and T. Gorschek. Extending value stream mapping through waste definition beyond customer perspective. *Journal of Software: Evolution and Process*, 26(12), 2014. 81, 93

[34] A. S. Lee and R. L. Baskerville. Generalizing generalizability in information systems research. *Information Systems Research*, 2003. 63

[35] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 2003. 47

[36] J. Liker. *The Toyota Way : 14 Management Principles from the World's Greatest Manufacturer*. McGraw-Hill Education, 2004. 79, 80

[37] S. McConnell. Managing technical debt. Technical report, Construx Software Builders, Inc, 2008. 54, 55, 85

[38] J. Meier, D. Hill, A. Homer, T. Jason, P. Bansode, L. Wall, R. Boucher Jr, and A. Bogawat. *Microsoft Application Architecture Guide*. Microsoft Press Book, 2009. 71

[39] S. Monsell. Task switching. *Trends in cognitive sciences*, 7(3), 2003. 89

[40] S. Mujtaba, R. Feldt, and K. Petersen. Waste and lead time reduction in a software product customization process with value stream maps. In *Proceedings of the 21st Australian Software Engineering Conference*, ASWEC, 2010. 81, 93

[41] B. Murphy. Code ownership-more complex to understand than research implies. *Software, IEEE*, 32(6):19, Nov 2015. 67

[42] T. Ohno. *Toyota production system: beyond large-scale production*. Productivity Press, 1988. 78, 79, 88

[43] G. Paci. Trucknumber. Portland Pattern Repository. URL: `http://c2.com/cgi/wiki?TruckNumberFixed`. 48

[44] J. L. Pierce, T. Kostova, and K. T. Dirks. Toward a theory of psychological ownership in organizations. *Academy of Management Review*, 26(2):298–310, 2001. 67

[45] M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, 2003. 78, 79

[46] M. Poppendieck and T. Poppendieck. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley Professional, 2006. 80, 90, 92

[47] K. Power and K. Conboy. Impediments to flow: Rethinking the lean concept of 'waste' in modern software development. In *Proceedings of Agile Processes in Software Engineering and Extreme Programming*, XP. Springer International Publishing, 2014. 81

[48] L. Prior. *Using documents in social research*. Sage, 2003. 25

[49] J. Rainsberger. Integration tests are a scam, 2013. URL: `https://www.youtube.com/watch?v=VDfX44fZoMc`. 60

[50] P. Ralph. Developing and evaluating software engineering process theories. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE, 2015. 56

[51] F. Ricca, A. Marchetto, and M. Torchiano. On the difficulty of computing the truck factor. In *Product-Focused Software Process Improvement*. Springer, 2011. 48

[52] 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. 48

[53] L. A. Rivera. Managing 'spoiled' national identity: War, tourism, and memory in croatia. *American Sociological Review*, 73(4):613–634, 2008. 30

[54] J. Saldana. *The Coding Manual for Qualitative Researchers*. Sage Publishing, 2012. 26

[55] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, 2001. 47

[56] 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. 54, 82, 98

[57] T. Sedano, P. Ralph, and C. Péraire. Sustainable software development through overlapping pair rotation. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement International Conference on Software Engineering*, ESEM, 2016. 89, 98

[58] T. Sedano, P. Ralph, and C. Péraire. Software development waste. In *Proceedings of the 2017 International Conference on Software Engineering*, ICSE '17, 2017. 98

[59] H. Selye. Stress without distress. In *Psychopathology of Human Adaptation*. Springer, 1976. 88

[60] S. Shingo and A. P. Dillon. *A study of the Toyota Production System: From an Industrial Engineering Viewpoint*. CRC Press, 1989. 78, 79

[61] W. Smale. The firm that starts work at 9.06am, 2016. URL: `http://www.bbc.com/news/business-37998577`. 15

[62] P. N. Stern. Eroding grounded theory. *Critical issues in qualitative research methods*, pages 212–223, 1994. 32

[63] 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. 19, 35, 36, 37, 62, 92

[64] A. Strauss and J. Corbin. *Basics of qualitative research*. Newbury Park, CA: Sage, 1988. 18, 19, 20, 26

[65] J. Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2), 1988. 87

[66] J. Sweller. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, 22(2), 2010. 87

[67] B. W. Tuckman. Developmental sequence in small groups. *Psychological bulletin*, 1965. 51

[68] A. Tweed and K. Charmaz. Grounded theory methods for mental health practitioners. *Qualitative Research Methods in Mental Health and Psychotherapy*, pages 131–146, 2011. viii, 17, 18

[69] J. J. Van Merriënboer and J. Sweller. Cognitive load theory and complex learning: Recent developments and future directions. *Educational Psychology Review*, 17(2), 2005. 87

[70] 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. 48

[71] VersionOne. 10th annual state of agile report, 2016. URL: `https://versionone.com/pdf/VersionOne-10th-Annual-State-of-Agile-Report.pdf`. viii, 12, 13

[72] D. Wells. Integrate often, 1999. URL: `http://www.extremeprogramming.org/rules/integrateoften.html`. 61

[73] M. Westman and D. Etzion. The impact of vacation and job stress on burnout and absenteeism. *Psychology & Health*, 16(5), 2001. 88

[74] E. Whitworth and R. Biddle. Motivation and cohesion in agile teams. In *Agile Processes in Software Engineering and Extreme Programming*. Springer, 2007. 73

[75] L. Williams and R. Kessler. *Pair Programming Illuminated*. Addison-Wesley Pearson Education, 2002. 48, 57

[76] J. P. Womack and D. T. Jones. *Lean thinking: banish waste and create wealth in your corporation*. Simon and Schuster, 1996. 78, 79, 80

[77] P. T. Yanos and K. Hopper. On 'false, collusive objectification': Becoming attuned to self-censorship, performance and interviewer biases in qualitative interviewing. *International Journal of Social Research Methodology*, 11(3):229–237, 2008. 24

[78] 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. 57