2017 IEEE/ACM 39th International Conference on Software Engineering

Software Development Waste

Todd Sedano
Pivotal
Palo Alto, CA, USA
Carnegie Mellon University
Silicon Valley Campus
Email: professor@gmail.com

Paul Ralph
University of Auckland
Auckland, New Zealand
University of British Columbia
Vancouver, BC, Canada
Email: paul@paulralph.name

Cécile Péraire
Carnegie Mellon University
Electrical and Computer Engineering
Silicon Valley Campus
Moffett Field, CA 94035, USA
Email: cecile.peraire@sv.cmu.edu

***Abstract*—*Context:* 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 of this paper 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 paper introduces the first empirical waste taxonomy. 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.**

***Keywords*-Software engineering waste, Extreme Programming, Lean Software Development**

I. 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 [1]. 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 [2], 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 [3], [4] 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 [5], [6]. 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 II summarizes the history of lean and review related work. Section III describes the research method. Section IV presents the emergent waste taxonomy. Section V compares this model with the waste list from Lean Software

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.20

TABLE I: 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 [1])	The cost of producing goods and services that do not meet the needs of the customer.
Non-utilized Talent (added by [7])	The cost of unused employee creativity and talent.

Development. Sections VI and VII evaluate the results, describe limitations, and conclude the paper.

II. A Brief History of Lean

Lean Thinking is a concept proposed by Womack [1] 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 [3], [4]. 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 [1]. 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 [4] shown in Table I. Later, Womack and Liker each added a waste type: value and non-utilized talent [1], [7].

Mary and Tom Poppendieck created Lean Software Development [5] by adapting Lean Thinking and the Toyota Production System from manufacturing to software development. Table II presents their comparison of manufacturing waste with software waste.

TABLE II: Comparison of Manufacturing Waste with Lean Software Development Waste

Toyota Production System's Manufacturing Wastes	Lean Software Development Wastes [8]
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 [1])	N/A
Non-utilized Talent (added by [7])	N/A

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.
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 [9].

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 [10], [11], [12]. One study identified the wastes of motion and extra processing from interviews, not the current state map [12]. These studies typically reduced waste by switching the organization from waterfall to iterative software development or reducing the batch size in iterative software development [10], [11], [12].

III. Research Method

We used Constructivist Grounded Theory [13], which involves iteratively collecting and analyzing data to generate and refine an emergent theory. We began by asking, "What is happening at Pivotal Labs when it comes to software development?" This led to the *Theory of Sustainable Software Development* [14] and two further core categories: *Team Code Ownership* [15] and *Removing Waste*, the topic of this paper.

Initially, the two primary data sources were participant observation field notes and interviews with Pivotal software engineers, interaction designers, and product managers. Interviews were recorded, transcribed, coded, and analyzed using constant comparison. The data advanced from initial codes to focused codes, focused codes to core categories.

When *Removing Waste* emerged as a core category, we incorporated data from retrospection meetings, performed additional interviews, and continued participant observation to refine the category. We constantly compared emerging findings to data from these three data sources until reaching saturation, as described below.

A. Research Context: Pivotal Labs

We selected Pivotal Labs as the research context because it is a successful software engineering organization, interested in using and evolving extreme programming, and open to research collaboration.

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 [16] in an environment conducive to agile development.

Typical teams include six software engineers, 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 practiced Extreme Programming [16] since the late 1990s. 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, refactoring, 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.

B. Data Collection

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

1) Participant Observation: The first author collected field notes while working 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.

2) Interviews: The first author interviewed 33 interaction designers, product managers, and software engineers who had experience with Pivotal's software development process from five different Pivotal offices. 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" [13]. 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.

The initial interviews began with the question, "Please draw on this sheet of paper your view of Pivotal's software development process." The interviewer intentionally avoided forcing initial topics. While exploring new emergent core categories, whenever possible, we initiated subsequent interviews with open-ended questions. The first author transcribed each interview with timecode stamps for each segment. These interviews were spread across the duration of the research study.

3) Retrospection Topics: When *removing waste* emerged as a core category, 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 [17]. 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 [17]. 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.

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

For co-located teams, the first author took a picture of the whiteboard at the end of the retro and later transcribed the topics 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.

C. Data Analysis

We began by iteratively collecting and analyzing field notes and interviews. We used line-by-line coding [13] 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 [18], 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.

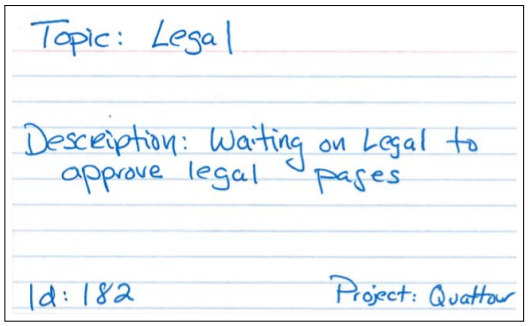

Fig. 1: Example Retro Topic Index Card

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 to facilitate reorganization. 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 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 2 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. The full chain of evidence is available [19].

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 removing waste in additional interviews and participant observations until no further waste-related categories were evident, i.e. theoretical saturation.

IV. Results: Types of Waste in Software Engineering

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

A. 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 [15], and customer satisfaction.

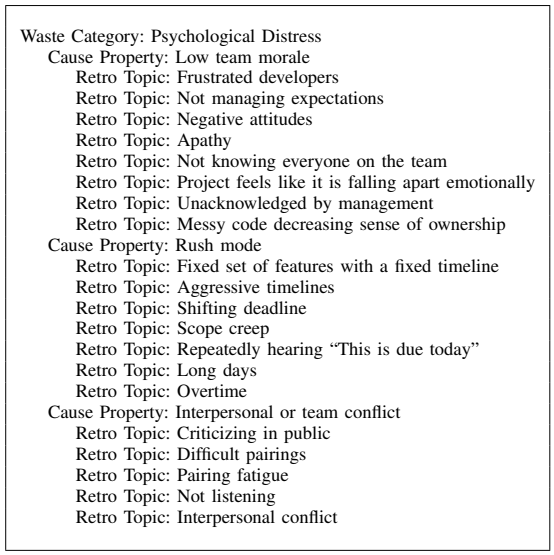

Fig. 2: Waste Organization Example (Psychological Distress)

The product features for Project Ses were designed based on a given persona—i.e. a fictional, archetypal user [20]. 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.

B. 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 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.

C. 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 mockups.

TABLE III: 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))

Technical debt refers to the risks of delaying needed technical work, by taking technical shortcuts, usually to meet a deadline [21]. 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

D. 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, some with expensive interactions to delight users.

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.

E. Waste: Extraneous Cognitive Load

Cognitive Load Theory [22] posits that our working memory is quite limited and overloading it inhibits learning and problem solving [23]. 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 [24]. Reducing the burden on working memory by removing extraneous cognitive load is therefore associated with more efficient learning among other positive outcomes [25].

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 reload 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).

F. Waste: Psychological Distress

"Stress is the nonspecific response of the body to any demand made upon it" [26]. 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 [27].

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.

G. 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 [3]. 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 [28]. For short waits, taking a break (e.g. playing table tennis) may be less wasteful than switching to another task.

H. 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 [14]. Teams promoting knowledge sharing appear less susceptible to knowledge loss from team churn or team member rotation.

I. 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 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.

TABLE IV: 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

V. Comparing to Lean Software Development

This section compares and contrasts our waste taxonomy (Section IV), with Lean Software Development's waste taxonomy [8] category by category.

While several categories are analogous (Table IV), we observed four types of waste not found in Lean: *unnecessarily complex solutions, extraneous cognitive load, psychological distress,* and *ineffective communication* (see Section IV 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 [8]. 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" [8], 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.

VI. 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 [13], [29]:

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 paper. 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 [10], [11], [12]. 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 [30]. We mitigated these risks by recording interviews and having the second and third authors review the coding process.

VII. Conclusion

This paper 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.

Several avenues for future research have potential. Studying different kinds of projects in different organizations may reveal new types of waste, or improve our understanding of existing categories. Furthermore, understanding wastes and their causes is just the first step in devising techniques for reducing waste and mitigating its effects. We would also like to investigate how teams use feedback loops to identify, manage, and reduce waste.

Acknowledgement

Thanks to Ben Christel for his assistance in sorting retro topics and helping with the initial analysis. Thank you to Rob Mee, David Goudreau, Ryan Richard, Zach Larson, Elisabeth Hendrickson, and Michael Schubert for making this research possible.

References

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

[2] 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*, vol. 25, no. 6, 2004.

[3] T. Ohno, *Toyota production system: beyond large-scale production*. Productivity Press, 1988.

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

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

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

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

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

[9] 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*, ser. XP. Springer International Publishing, 2014.

[10] 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.

[11] M. Khurum, K. Petersen, and T. Gorschek, "Extending value stream mapping through waste definition beyond customer perspective," *Journal of Software: Evolution and Process*, vol. 26, no. 12, 2014.

[12] 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*, ser. ASWEC, 2010.

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

[14] 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*, ser. ESEM, 2016.

[15] ——, "Practice and perception of team code ownership," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE, 2016.

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

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

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

[19] T. Sedano, "Sustainable software development: Evolving extreme programming," Ph.D. dissertation, Carnegie Mellon University, 2017.

[20] J. Grudin and J. Pruitt, "Personas, participatory design and product development: An infrastructure for engagement," in *Participatory Design Conference*, 2002.

[21] S. McConnell, "Managing technical debt," Construx Software Builders, Inc, Tech. Rep., 2008.

[22] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cognitive Science*, vol. 12, no. 2, 1988.

[23] A. R. Artino Jr, "Cognitive load theory and the role of learner experience: An abbreviated review for educational practitioners," *Aace Journal*, vol. 16, no. 4, 2008.

[24] J. Sweller, "Element interactivity and intrinsic, extraneous, and germane cognitive load," *Educational Psychology Review*, vol. 22, no. 2, 2010.

[25] J. J. Van Merriënboer and J. Sweller, "Cognitive load theory and complex learning: Recent developments and future directions," *Educational Psychology Review*, vol. 17, no. 2, 2005.

[26] H. Selye, "Stress without distress," in *Psychopathology of Human Adaptation*. Springer, 1976.

[27] M. Westman and D. Etzion, "The impact of vacation and job stress on burnout and absenteeism," *Psychology & Health*, vol. 16, no. 5, 2001.

[28] S. Monsell, "Task switching," *Trends in cognitive sciences*, vol. 7, no. 3, 2003.

[29] 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*, ser. ICSE, 2016.

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