CHAPTER 19

Removing Software Development Waste to Improve Productivity

Todd Sedano, Pivotal, USA
Paul Ralph, Dalhousie University, Canada
Cécile Péraire, Carnegie Mellon University Silicon Valley, USA

Introduction

As we have seen in previous chapters, measuring the productivity of software professionals is challenging and hazardous. However, we do not need sophisticated productivity measures to recognize when time and effort are wasted. When we see software engineers rewriting code because the previous version was hastily done, their productivity is obviously suffering.

In project management, *waste* refers to any object, property, condition, activity, or process that consumes resources without benefiting any project stakeholder. Waste in a development process is analogous to friction in a physical process—reducing waste improves efficiency and productivity by definition.

However, reducing waste can be challenging. Waste is often hidden by bureaucracy, multitasking, poor prioritization, and invisible cognitive processes. People quickly acclimate to wasteful practices—*that's just how we do things here*. The actions necessary in tackling wastes are waste *prevention*, *identification*, and *removal*. Those actions require us to understand the kinds of waste present in software projects.

C. Sadowski and T. Zimmermann (eds.), *Rethinking Productivity in Software Engineering*,
https://doi.org/10.1007/978-1-4842-4221-6_19

To better understand software development waste, we conducted an extended participant-observation grounded theory study at Pivotal Software. Pivotal is a large American software development organization, known for using and evolving extreme programming [1]. Pivotal builds software products and provides agile transformation services for its clients.

Grounded theory is a research method for systematically generating scientific explanations from empirical data. Participant-observation is a type of data collection in which the researcher takes part in the project to gain an insider's perspective. We observed Pivotal teams working on agile transformation projects with engineers from Pivotal's clients in various domains. The study involved two years and five months of participant-observation, 33 intensive open-ended interviews, and one year's worth of retrospection data. It is the first empirical study of waste in software development. For more information about the research method, see Sedano et al. [7].

Taxonomy of Software Development Waste

During the study, we observed nine types of waste (Figure 19-1). This section explains each waste type and associated tensions that complicate reducing the waste.

Figure 19-1. *Types of Software Development Waste (© Todd Sedano)*

Building the Wrong Feature or Product

The cost of building a feature or product that does not address user or business needs.

One of the most serious types of waste is building features that no one wants or needs. A more extreme version is building an entire product that no one wants or needs.

For example, on one Pivotal team, three engineers spent three years building a system without ever talking to potential users. The delivered system did not fulfill the users' needs. After spending nine months trying to alter the system to meet user's needs, management scrapped the project. Another example involved building a healthcare relationship management system. During user-centered design, the team ignored user feedback. After a year of trying to find people who would use the delivered system, they ran out of money.

We observed two main causes of "building the wrong feature or product":

- *Ignoring user desiderata*: This includes not doing user research, validation, or testing; ignoring user feedback; and working on features with low user value.
- *Ignoring business desiderata*: This includes not involving a business stakeholder, slow stakeholder feedback, and unclear product priorities.

Techniques for avoiding or reducing this waste include:

- Usability testing
- Feature validation
- Frequent releases
- Participatory design

Building the wrong features or products appears related to a specific tension: user versus business needs. In other words, sometimes users' needs conflict with business needs. For example, for one mobile application, the marketing organization insisted on including the company news feed. Users did not want the news feed and perceived it as spam, lowering their opinion of the mobile application.

Mismanaging the Backlog

> *The cost of duplicating work, expediting lower value user features, or delaying necessary bug fixes.*

One kind of prioritization problem specific to agile software development is *backlog inversion*. In principle, all of the stories are kept in a prioritized backlog such that whatever is on top of the backlog is what the product manager (or equivalent) wants done next. In practice, however, some product managers only prioritize the top *n* stories, after which is a jumble of medium-priority, low-priority, and outdated stories. Backlog inversion occurs when the team gets ahead of the product manager and starts working on story *n+1*.

For instance, on Monday, the product manager examines the backlog and re-prioritizes the next seven stories. The team finishes those seven stories and begins working on stories eight, nine, and ten. Since these stories have not been prioritized recently, the team might unknowingly be working on low-priority stories.

Mismanaging the backlog includes all the waste associated with poor prioritization. We observed numerous causes of "mismanaging the backlog" waste:

- Backlog inversion
- Working on too many features simultaneously
- Duplicated work
- Not enough ready stories
- Imbalance between feature work and bug fixing
- Delaying testing or critical bug fixing
- Capricious thrashing (see below)

Solutions for avoiding or reducing this waste include:

- Prioritizing the backlog several times a week
- Minimizing work in progress by finishing features before starting new ones
- Updating the backlog with current work in progress
- Writing enough stories to stay ahead of development
- Routinely working on bug fixes while doing feature development
- Receiving feedback from users before making changes

This waste is also related to a tension: intransigence versus capricious trashing. 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). As an example of trashing, on one project, the launch was delayed while the business fiddled with the sequence and number of steps in the user registration process.

Rework

The cost of altering delivered work that should have been done correctly but was not.

Not all rework is waste. Wasteful rework refers to the cost of altering delivered work *that should have been done correctly but was not*. Reworking a product because of unforeseeable or unpredictable circumstances is not waste.

For example, one enterprise team had been shipping Python code while accumulating technical debt over time. The code became so unmanageable that they decided to re-write it in Go from scratch. We see the entire rewrite as rework because ignoring technical debt impairs the understandability and modifiability of software over time, and the team could have avoided the rework by refactoring the original Python code before it became unmanageable.

We observed the following causes of "rework" waste:

- Technical debt, that is, technical work delayed by taking shortcuts to save time and meet deadlines.
- Ambiguous story definition, including ambiguous acceptance criteria and mock-ups.
- Rejected stories, that is, when a product manager rejects a story implementation because it does not satisfy the acceptance criteria.
- Defects, including poor testing strategy and not performing root-cause analysis on defects.

Solutions for avoiding or reducing this waste include:

- Continuous refactoring
- Reviewing acceptance criteria before beginning a story

- Verifying acceptance criteria before finishing a story
- Improving testing strategy and root-cause analysis on bugs

Refactoring code to handle new features is not waste. A team cannot anticipate and predict future work to be done. Instead, we recommend teams focus on aligning their code with their current understanding of the system features and code design. A team that routinely refactors its code reduces onboarding developer costs and increases its ability to deliver new functionality. Clean code has additional benefits: it is easier to understand, easier to modify, and has fewer defects. Refactoring code to support new functionality is part of the inherent cost of the new functionality. In contrast, rushing a feature introduces technical debt, which leads to rework and extraneous cognitive load.

Rework waste is related to a ubiquitous tension between doing things well and doing things quickly. A recent study of decision-making during programming found that this tension affects many developer actions, including whether to refactor problematic code and whether to implement the first approach that comes to mind or research better ones [5].

Unnecessarily Complicated or Complex Solutions

> *The cost of creating a more complicated solution than necessary; a missed opportunity to simplify features, user interface, or code.*

Unnecessary complexity is intrinsically wasteful and harmful [3]. The more complicated a system is, the more difficult it is to learn, use, maintain, extend, and debug.

Unnecessary feature complexity wastes users' time as they struggle to understand how to use the system and achieve their objectives. For instance, one product required the user to fill in form fields not related to the task at hand. Implementing and maintaining those unnecessary fields is a waste of developer time and an opportunity to introduce defects.

We observed the following causes of "unnecessarily complicated or complex solutions" waste:

- Unnecessary feature complexity from the user's perspective. This includes overly complex user interactions and business processes.

- Unnecessary technical complexity from the team's perspective. This includes duplicating code, lack of interaction design reuse, and overly complex technical design.

Solutions for avoiding or reducing this waste include:

- Prefer simpler designs for user interaction
- Prefer simpler designs for software code
- Consider whether each proposed feature is worth the additional complexity it will introduce

We observed the following tension in relation to this waste: big design up-front versus incremental design. Up-front designs can be based on incorrect or out-of-date assumptions, leading to expensive rework especially in rapidly changing circumstances. However, rushing into implementation can produce ineffective emergent designs, also leading to rework. Despite the emphasis on responsiveness in agile development, designers struggle to backtrack on important decisions and features [2].

The logic of avoiding rework underlies disagreement over big design up-front versus incremental design—proponents of both approaches feel that they are reducing rework. However, on the observed projects, no amount of up-front consideration appears sufficient to predict user feedback and product direction. Therefore, the observed teams preferred to incrementally deliver functionality and delay integrating with technologies until a feature required it.

Extraneous Cognitive Load

The costs of unnecessary mental effort.

Human beings have limited working memory and mental resources. Technically, cognitive load refers to how much working memory a task requires. Here, however, we are using extraneous cognitive load more generally to mean the costs of making something unnecessarily mentally taxing.

For example, one project used five separate test suites that each worked differently. Running the tests, detecting failures, and rerunning just a failed test required learning five different systems. This was unnecessarily cognitively taxing in two senses: developers had to learn the five systems initially, and developers had to remember how all five systems worked and avoid confusing them.

We observed the following causes of "extraneous cognitive load" waste:

- Technical debt
- Complex or large stories
- Inefficient tools and problematic APIs, libraries, and frameworks
- Unnecessary context switching
- Inefficient development flow
- Poorly organized code

Solutions for avoiding or reducing this waste include:

- Refactor code that is difficult to understand
- Decompose large, complex stories into smaller, simpler stories
- Replace hard-to-use libraries
- Work on one task at a time until it is completed; avoid "blocking" tasks (i.e., putting a task on hold to work on something else)
- Improve the development flow including better scripts and tools

Psychological Distress

The costs of burdening the team with unhelpful stress.

Stress can be beneficial ("eustress") or harmful ("distress"). For instance, a little pressure from knowing that the client has high expectations can motivate a team to deliver a better product. Contrastingly, worrying about a sick family member, being yelled at by an angry client, or thinking you might lose your job can reduce performance.

Psychological distress can be either harmful stress or just too much stress. How much stress is too much depends on the person, but everyone has a limit after which more stress lowers performance. Both distress or extreme stress are distracting and draining. Stress can make people feel anxious, overwhelmed, and unmotivated. Therefore, we see psychological distress as intrinsically wasteful.

For example, we observed stress resulting from snarky remarks about other teams or other developers on mailing lists, including "Wow! 22 commits with zero pull requests there." Another example was a countdown to a release date written on an office

whiteboard. The team felt that over-emphasizing the deadline was increasing stress and leading to poor technical decisions. Eventually, the countdown was erased from the whiteboard.

Different people find different experiences distressing. However, some common distress-inducing experiences we have observed include:

- Low team morale
- Rush mode
- Interpersonal or team conflict
- Inter-team conflict

A wealth of research investigates the nature, causes, and effects of stress. A full treatment of stress in software engineering would fill a large book. The present study, in contrast, supports only a few basic recommendations for detecting and reducing stress.

- In our experience, detecting distress is not difficult—simply asking team members, "How are things going?" is usually sufficient.
- Stress related to deadlines can sometimes be mitigated by reducing scope or extending the deadline.
- Stress related to interpersonal conflict can be mitigated by facilitated mediation.

Knowledge Loss

The cost of re-acquiring information that the team once knew.

A team can lose knowledge when a person with unique knowledge leaves, when an artifact containing unique knowledge is lost, or when the knowledge is sequestered within one person, group or system. Regardless of how the knowledge was lost, the cost of re-acquiring it is a type of waste.

We observed the following causes of "knowledge loss" waste:

- Team churn (that is, staff rotating on and off a team)
- Knowledge silos (that is, where important information is sequestered within one person, group or system)

In Sedano et al. [6], we propose several practices for encouraging knowledge sharing and continuity including continuous pair programming, overlapping pair rotation, and knowledge pollination (e.g., stand-up meetings). Although we have not observed it directly, code review may also help knowledge sharing and prevent knowledge loss.

This waste is related to the tension between sharing knowledge through interaction vs. documentation. One of the key insights of the agile literature is that sharing knowledge face-to-face is usually more effective than sharing knowledge through written documents. Indeed, often documentation quickly becomes outdated and unreliable.

Waiting/Multitasking

The cost of idle time, often hidden by multitasking.

When something goes wrong in a manufacturing plant, we can sometimes see people waiting around. If the boxing team runs out of boxes, they might just stand idle until more boxes arrive. This is obviously waste.

Waiting waste is less obvious among software professionals because waiting is often hidden by multitasking. For example, if the integration process takes an hour, programmers tend to switch to some other, lower-priority work while waiting for integration.

We observed the following causes of "waiting/multitasking" waste:

- Slow or unreliable tests
- Missing information, people, or equipment
- Product managers taking too long to provide needed information
- Context switching between tasks

Solutions for avoiding or reducing this waste include:

- Expose waiting time by limiting work in progress
- For short waits, take breaks (e.g., play table tennis) instead of task switching
- For longer waits, use waiting time to work on the cause of the wait (e.g., shorten a long build)

Multitasking introduces waste in two ways. First, multitasking involves a mental transition to the new task, which can be quite time-consuming, especially if the new task is cognitively demanding. Second, multitasking creates dilemmas when the original high-priority task becomes available again. Do developers finish the second lower-priority task (delaying higher priority work) or immediately switch back to the original task (leaving work-in-progress)?

Engineers remaining idle for more than a few minutes is typically viewed negatively. Thus, engineers tend to prefer context-switching over waiting despite the drawbacks described above.

Ineffective Communication

> *The cost of incomplete, incorrect, misleading, inefficient, or absent communication among project stakeholders.*

Ineffective communication is intrinsically wasteful. For example, a product manager notices a bug and adds it to the backlog but does not explain how to reproduce it. The team ends up sleuthing—either experimenting with different possible combinations or asking the product manager for additional details. As another example, a developer changes key configuration information that affects all other developers on the team. Instead of telling everyone that they need to pull the latest code, the developer posts about the change via asynchronous communication (e.g., Slack). Some developers do not see this communication and wonder why their code stops working. They waste time trying to figure out the solution when the answer was already known within the team.

We observed the following causes of "ineffective communication" waste:

- Teams that are too large.
- Asynchronous communication, which is especially problematic for distributed teams, distributed stakeholders, and when the team depends on other teams or opaque processes outside the team.
- One person or a few people dominating the conversation or not listening.
- Inefficient meetings including lack of focus during meetings, skipping retros, not discussing blockers each day, and meetings running over (e.g. long stand-ups).

Like stress, copious research has investigated communication effectiveness, and a complete account is beyond the scope of this chapter. However, we can make some simple recommendations.

- Synchronous (especially face-to-face) communication seems more effective for most people, most of the time.
- Conversational turn-taking, where participants take turns speaking one at a time, leads to better shared understanding.
- More powerful participants (e.g., white male project manager) interrupting less powerful participants (e.g., nonwhite female junior developer) has a chilling effect on diversity of thought and quality of group decision-making. Other participants can mitigate interruptions by returning to the interrupted speaker by, for example, saying "Can we come back to what Alexis was saying about…."

Ineffective communication might lead to the other types of waste. For instance, ineffective communication resulting in delays might lead to the waiting waste. Ineffective communication resulting in misunderstanding user or business needs might lead to building the wrong feature or product, or misunderstanding the existing solution might lead to building an overly complex solution and extraneous cognitive load. Ineffective communication resulting in poor decision-making might lead to mismanaging the backlog. Ineffective communication resulting in technical mistakes might lead to defects and rework. Ineffective communication resulting in misunderstandings among team members might lead to conflicts and psychological distress. These are just a few examples highlighting the importance of effective communication and how poor communication can generate waste.

Additional Wastes in Pre-agile Projects

Since Pivotal is lean and agile, it has already eliminated some common types of waste. Professionals using waterfall, plan-driven, or other pre-agile approaches may experience waste from unnecessary bureaucracy. Some bureaucracy is necessary to govern

(especially large) organizations. However, much bureaucracy is simply pointless, and some is actively harmful. Examples include:

- *Overplanning*: This involves estimating budgets, schedules, phases, milestones, or tasks at a level of detail that is not supported by the information at hand or the stability of the project environment. When a plan requires copious guesses and assumptions, it is a fantasy, not a plan. Overplanning not only wastes the planner's time but also engenders psychological distress when reality departs from the plan.
- *Overspecifying*: This involves specifying requirements or design at a level of detail that is not supported by the information at hand. Overspecifying is a common problem in projects with large, up-front requirements and design phases. Warning signs include copious optional, low-priority, or low-confidence requirements; developing an elaborate architecture while stakeholders are still arguing about the goals of the project; fleshing out features that will not be built for months, if ever. Overspecification is not only a waste of time, it can constrain developers, obscure better solutions, and reduce creativity.
- *Performance metrics*: Perhaps the main theme to emerge from the study of performance measurement is that measuring performance reduces performance. All metrics can be gamed, and gaming metrics is distracting and time-consuming. Measuring people just motivates them to engage in metric-optimizing theatrics, which are usually less efficient than what they were doing before the metrics. Attempts to quantify performance are therefore not just wasteful but often counterproductive, especially where bonuses are tied to the measurements [4].
- *Pointless documentation*: Some documentation is necessary—even critical—when it helps achieve a specific goal. However, some projects have binders full of documentation that will not be read before growing out-of-date, if ever. Pointless documentation is a form of *ineffective communication* waste.

- *Process waste*: Processes can be wasteful when they generate pointless documentation (reports, forms, formal requests), pointless meetings (like large company or department-wide meetings, not team meetings), pointless approvals (due to not trusting the people who do the work), and handoffs.
- *Handoffs*: Organizations that divide projects into phases and have different teams involved in different phases of the same project experience handoff waste. Handoff waste is the cost (in knowledge, time, resources, and momentum) of passing a project from one team to another. Handoffs contribute to other wastes including knowledge loss, ineffective communication, and waiting.

When following pre-agile practices, two general strategies may help reduce waste. First, hunt for slow-feedback loops, as shortening feedback loops often helps to reduce waste. Second, actively remove the policies responsible for the waste. One problem with bureaucracy is that, once a policy is made, following the policy becomes the bureaucrat's goal, regardless of the organizational goals the policy was written to support. Waste is the inevitable byproduct of optimal actions for achieving organizational goals diverging from the actions prescribed by flawed or outdated policies.

Discussion

The above discussion may appear to suggest that all problems are types of waste, but that is not the case. This section discusses what is special about waste, and gives more suggestions for removing waste.

Not All Problems Are Wastes

It is tempting but incorrect to label anything that goes wrong on a project as waste. Human beings make mistakes. A developer may accidentally push code before running the test suite. Our knowledge is limited. A product manager may write an impractical user story because he or she does not know of some particular limitation. We forget. A developer might forget that adding a new type to the system necessitates modifying a configuration file. Whether we conceptualize these sorts of errors as waste is a matter of opinion, but focusing on them is unhelpful because they are often unpredictable.

It is better to focus on *systemic* waste: waste that affects a wide variety of projects in consistent, predictable, and preventable ways.

Similarly, it is important to distinguish foreseeable errors from actions that only seem like errors in hindsight. Suppose that users clearly indicate that a particular feature is not desirable, but we build it anyway, and sure enough, no one uses the feature. Obviously, this is waste. In contrast, suppose users are clamoring for a feature, so we build it, but it's quickly abandoned as users realize it does not really work for them. This is not an error; it's learning. Sometimes, building a feature, prioritizing the wrong thing, refactoring, and communicating badly are the only ways of learning what is actually needed. The concept of waste should not be misused to demonize incremental development and learning.

Reducing Waste

Reducing waste is often straightforward. The countdown on the whiteboard is stressing out the team? Erase it. Five separate test suites take forever to run? Integrate them. Building a feature no one has asked for? Stop. User interface is too complex? Simplify it. Not enough knowledge sharing among programmers? Pair-program. The official approval process is inefficient? Change it. Sometimes this is easier said than done, but it's not rocket science either.

The problem is that waste is often hidden. Rework is hidden in "new features" and "bug fixes." Building the wrong features is hidden by lack of good feedback. Knowledge loss is hidden by not realizing the organization used to know this information. We hide distress to avoid looking weak. Bureaucracy hides waste behind an official policy. That is why this chapter describes all different sorts of waste—waste is easier to identify if you know what to look for.

Once we have identified some waste, there are three broad approaches for reducing it: prevention, incremental improvement, and "garbage day":

- *Prevention*: This involves creating systems that impede waste. User research impedes "building the wrong feature" waste. Continuous refactoring impedes "rework" waste. Pair programming, peer code review, and overlapping pair rotation impede "knowledge loss" [6]. Daily stand-ups impede "inefficient communication" waste.

- *Incremental improvement*: Waste reduction can be approached as a continuous improvement practice, running parallel to feature development. Waste reduction can be discussed in retrospective meetings, and one or two waste reduction tasks can be included in the backlog each week. This is a good approach for most teams, since suspending development for weeks to remove waste is not tenable in most organizations and could reduce team morale and customer satisfaction.
- *Focused waste reduction: garbage day/trash pickup day*: Some companies set aside special periods where employees are free to work autonomously. For example, Pivotal has a "hack day" during which employees can work on a theme or whatever they want. Organizations can implement a similar set period ("garbage day") in which employees tackle some source of waste, for instance, speeding up the integration process, removing redundant tests, simplifying an overcomplicated process, or just meeting with co-workers to share siloed knowledge.

A related question is, "If we have identified several different kinds of waste, what should we tackle first?" We observed teams prioritizing waste removal using the following procedure:

1. Individually list several wastes.
2. Plot each waste on a graph like Figure 19-2.
3. Prioritize wastes beginning with the best ratio of easy to remove and high impact (e.g., W1) and working your way down to wastes that are harder to remove and have less impact (e.g., W8).
4. Add waste reduction to the backlog (as chores) and prioritize these chores as time permits.

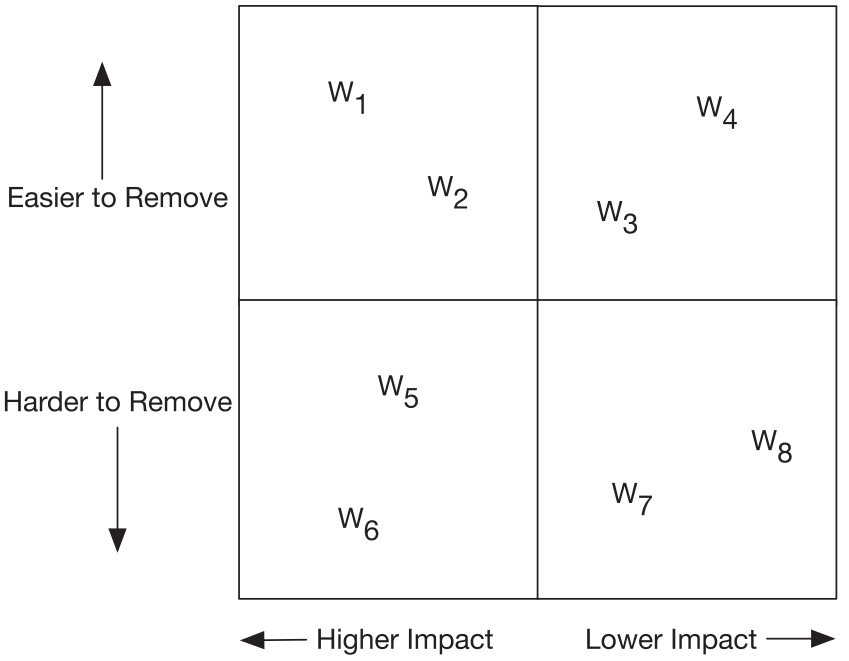

Figure 19-2. *Prioritizing waste removal*

Of course, eliminating some (low impact, hard-to-remove) wastes may not be worth the cost. For example, having a distributed team most often contributes to ineffective communication waste, but it might be the most practical solution when experts with rare skills are distributed across the globe. Eliminating waste should be and typically is a secondary goal. Waste elimination should not displace the primary goal of delivering a quality product.

Here, we recommend prioritizing wastes based on our best guesses as to their impact. Precisely quantifying the impact of each waste is impractical. How would you quantify the inefficiencies of overburdening developers with unhelpful stress and the impact on their health, or the impact of knowledge loss, when the team does not even know what knowledge is being lost? Quantifying waste might be a good PhD project but is likely not worth the trouble for most professional teams.

Conclusion

In summary, software *waste* refers to project elements (objects, properties, conditions, activities, or processes) that consume resources without producing benefits. Wastes are like friction in the development process. An important step in tackling this friction is waste awareness and identification. During our study, we identified nine main types of waste in agile software projects: 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. For each waste type, we proposed some suggestions to reduce the waste. Reducing wastes removes friction and hence improves productivity.

Software professionals have become increasingly focused on productivity (or *velocity*), often leading to increasingly risky behavior. Moving as fast as possible is great until someone quits, gets sick, or goes on vacation and the team suddenly realizes that no one else knows how a large chunk of the system works or why it was built that way. For many companies, stability and predictability are more important than raw speed. Most firms need software teams that steadily deliver value, week after week and month after month, despite unexpected problems, disruptions, and challenges.

Eliminating waste is just one way to forge more resilient, disruption-proof teams. This work on waste is part of a larger study of sustainability and collaboration in software projects. In Sedano et al. [6], we propose a theory of sustainable software development that extends and refines our understanding of extreme programming with new, sustainability-focused principles, policies, and practices. The principles include engendering a positive attitude toward team disruption, encouraging knowledge sharing and continuity, and caring about code quality. The policies include team code ownership, shared schedule, and avoiding technical debt. The practices include continuous pair programming, overlapping pair rotation, knowledge pollination, test-driven development, and continuous refactoring.

Based on our experiences, none of the results presented in this chapter appears unique to Pivotal Software or extreme programming. However, our research method does not support statistical generalization to contexts beyond the observed teams at Pivotal Software. Therefore, researchers and professionals should adapt our findings and recommendations to their own contexts, case by case.

Key Ideas

The following are the key ideas from this chapter:

- There are several different types of preventable "wastes" that occur during software development and represent lost productivity.
- While it may be hard to define and measure productivity, identifying/reducing waste is an effective way to become more productive.

References

[1] Kent Beck and Cynthia Andres. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, 2004.

[2] Nigel Cross. Design cognition: results from protocol and other empirical studies of design activity. In Design knowing and learning: Cognition in design education. C. Eastman, W.C. Newstetter, and M. McCracken, eds. Elsevier Science. 79–103. 2001.

[3] John Maeda. The Laws of Simplicity. MIT Press. 2006.

[4] Jerry Muller. The Tyranny of Metrics. Princeton University Press. 2018.

[5] Paul Ralph and Ewan Tempero. Characteristics of decision-making during coding. In Proceedings of the International Conference on Evaluation and Assessment in Software Engineering, 2016.

[6] Todd Sedano, Paul Ralph, and Cécile Péraire. Sustainable software development through overlapping pair rotation. In Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2016.

[7] Todd Sedano, Paul Ralph, and Cécile Péraire. Software development waste. In Proceedings of the 2017 International Conference on Software Engineering, 2017.