Refactoring with Observability: Novel Practices for Learning

DAVID LARIBEE, Kata.io

Best practices imply a vision for how a team might behave or employ tools. It's only by engaging in real work that we find how those practices fit our actual context. This paper details the origin story of two novel practices, Refactoring with Telemetry and Architectural Mapping, and the process by which we discovered them, after finding an impedance mismatch between the practice as advertised and the realities presented by larger, more complex systems.

1. INTRODUCTION

In a perfect world, everyone on your team would have a clear representation of the systems they work on in their head. If you asked engineers to come up to a whiteboard and draw a picture of their software architecture, they'd each produce a clear, lucid diagram in a matter of minutes. Taking a step back, we'd see each individual's picture would resemble their neighbor's: evidence that the team shares a single, cohesive mental model about how their system is organized and operated.

This scenario describes the rosy worldview of "best practices." It is an aspiration or vision gained only through deliberate practice and conscious discipline. I've had a problem with the term "best practice" for a long time now because they seem to frame an alternative workflow as a promised land, a panacea. Between the starting point of selecting a practice and the destination of deriving value from the method, there's often a profound journey of adaptation and understanding. What you end up doing might even be a new practice altogether.

As a coach and consultant, this conflict comes up in a lot of the client work I do. Listening to this friction and channeling it into a better, more-productive learning experience is a big part of the job for me. Sometimes this leads me and the teams I'm coaching into new territory and onto new ideas for how our practice should change depending on our context.

This paper details the origin of two novel practices that help teams effectively refactor large, complex software systems: Refactoring with Telemetry and Architectural Mapping.

Refactoring with telemetry establishes a tight feedback loop between structural changes a development team makes and sometimes esoteric code quality metrics. It's an easy-to-use learning tool that makes clear the close relationship between small, continuous refactoring and long-term sustainability.

To map architectures and previsualize changes, we use the C4 Model of software architecture. A subset of UML, C4 turns diagrams into useful maps relevant for different audiences and contexts (business stakeholders, ops, and dev staff). I'll share how we engage teams in the mapping process and use the maps to align technical debt repayment with the delivery of value, invite multiple perspectives and new ideas, and engender shared mental models around larger legacy systems. Think of it as story mapping for systems.

I'll share the mechanics of these practices, giving you an idea of how to refine your own practice experiments using our results as a point of acceleration. More importantly, I'll reflect on the starting conditions required to be in place to create and discover your own new, more relevant practices. I'll challenge the idea that best practices are a universal truth; their real value is in creating a starting point to find ideas you can incorporate into your own workflow. It's not the destination. It's the journey.

2. BACKGROUND

One of the practices I teach is Test Driven Development (TDD). TDD is very much about achieving and maintaining a state of simplicity and minimalism in your design: just enough code to satisfy the delivery requirement and not a line more. This state helps us introduce new changes safely as we learn about the problem we're solving.

David Laribee, Atlanta USA; email: david@kata.io

What happens when you show up to teach a 2-day class on how to practice TDD with 20 adult learners who work on heavily coupled legacy systems with high cognitive load? In our controlled classroom environment, we can happily and productively test-drive sample exercises all day long, but what happens when teams return to their day job where they find it impossible to write even a single isolated test?

When I started spending time with these teams in a coaching capacity, an epiphany came to me after the class concluded. What I discovered was that TDD, as a best practice, did not apply to this group's context. At most, TDD was an aspiration. Rather than starting by writing a failing test, they needed to lead with refactoring and tidying their codebase so they might maneuver with a design practice such as TDD. This group needed a playbook that was much more context-specific than any book about or class on TDD could provide.

The Cynefin framework [Snowden] helps in reasoning about domain complexity and what kinds of practices we might expect to be successful, given our context (see Figure 1 - The Cynefin Framework).

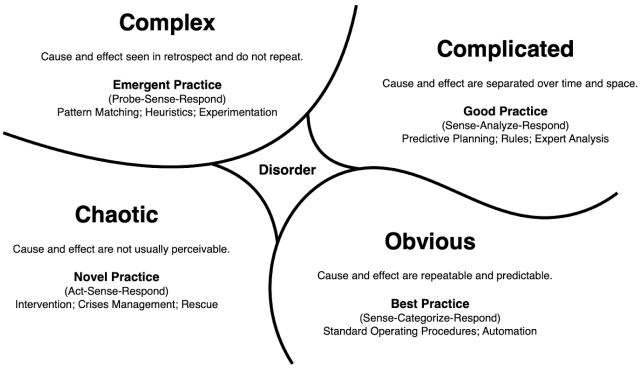

Figure 1: The Cynefin Framework

The group I described working with above was probably operating in the "complex" or "chaotic" domains. In this domain, we have to tend to change our systems through a series of small, safe-to-fail experiments (complex) or by reacting to interpretations and correlations of events (chaotic). TDD, on the other hand, as a best practice, makes a lot of sense in the "obvious" domain. As a design methodology, it's going to work a lot better when you begin to design. As is often said, TDD is a whole lot easier to practice and understand when we're starting from scratch, on a greenfield product or a 2-day training class.

Cynefin (Figure 1) tells us that if we're operating from "complex" or "chaotic," we might need to look for practices that are emerging or invent our own, so-called "novel practices." As teachers and coaches, we probably need to look for a different venue to operate from this understanding. Classrooms are not the right setting. Pre-planned exercises often lose relevance when it comes time to implement the practice.

Since 2015, we've been experimenting with immersive learning environments called "dojos." I was part of an initiative at a large retailer where we'd take teams on a 6-week journey of learning centered on their authentic backlog and real systems. It's a radical departure from the standard model of organizational learning: training classes, certifications, hire an army of coaches, rinse-and-repeat transformation, etc.

This model, or something like it, is an essential factor in getting to a place where practices can be invented (novel) and refined (emerge). For each practice, Refactoring with Telemetry and Architectural Mapping, I'll describe the moment where pattern matching and team-coach collaboration led us collectively to something that didn't resemble a well-known best practice. Let's start with Refactoring with Telemetry.

3. REFACTORING WITH TELEMETRY

Refactoring, as defined by Martin Fowler [Fowler], is "changing the structure of code without changing its behavior." Structuring complex code into more generalized solutions that are easier to comprehend and maintain is also an essential step in the TDD loop:

1. Red - write a failing test.
2. Green - write just enough code to make the test past.
3. Clean - refactor your code to a cleaner, more comprehensible state.
4. Repeat

The TDD loop encounters disharmony when working with existing code that not designed in this manner. The first problem engineers are likely to encounter when adapting TDD to a monolithic legacy system is how to get existing code under test. Often their first attempts fail; large swathes of the system, including attached resources and external dependencies, get pulled into a test harness. Isolating and specifying small behaviors is impossible and frustrating when you have to deal with the whole world.

3.1 Adapting & Unbundling

After the class and my epiphany, I found myself with a couple of teams in front of a large monitor with access to their actual code, backlog, documentation, and other supporting artifacts. Fresh off of a 2-day TDD course, we hunkered down in a large conference room and discussed both the practicality and relevance of what they had just learned. The same concerns kept coming up, best summarized as: how do we start writing tests first when you cannot isolate a module, class, or function?

This recognition led me to suggest we start the Red-Green-Clean loop at the clean step. Why not find some refactoring we could do to get a chunk of the code into isolation and make new stories impacting that code much easier to get into a test harness? In effect, we unbundled the sub-practice of refactoring from the parent practice of TDD. We're still in the territory of, if not best, "good practice." In Cynefin terms, we're traveling from the domain of order where TDD lives (remember TDD works best when we start with that design approach) toward the team's state of "complex" where we're on the lookout for a novel or emerging practice. It's time to improvise and adapt.

3.2 Where do we start?

Once we decided on a tactic for getting started, the next question prompted itself: what should we start refactoring? The group chose to hunt for a juicy refactoring target the majority of people would encounter. It's important to note that this choice would be a lucky turn in our collective discovery of a new workflow, a new practice. You'll see why in a minute.

We had access to a static analysis tool for bringing some objectivity to our code quality discussions. In this case, we had NDepend [NDepend] available to us, a powerful tool that yields all kinds of insight into what's happening in a codebase as a whole. Without going into extreme detail, NDepend makes it easy to find large, complex classes that make for high-value targets of refactoring opportunity (see Figure 2: NDepend: Metrics View).

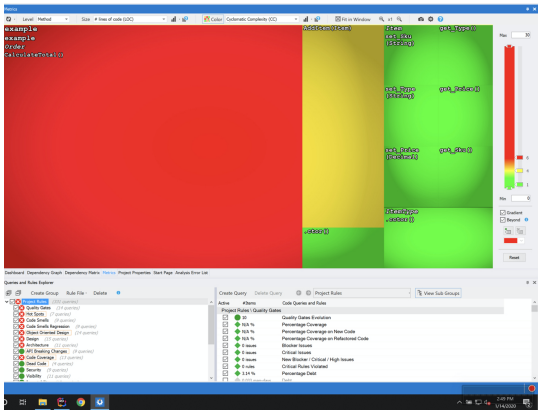

Figure 2 – NDepend Metrics View

NDepend's "Metrics View" presents a heatmap of color-coded squares. The smaller squares each represent a class in the system. The color of the square indicates the class's total complexity, with red exceeding a user-defined threshold. Complexity tells us how many execution paths through a class there are. As a metric, it's a directly correlated leading indicator of cognitive load. High complexity in classes might mean code that's hard to understand. The area of each square indicates the number of lines of code. Large, red boxes tend to be hotspots where there's probably some level of implementation debt accumulating. These often present an excellent prospect for a refactoring mission.

Having found our target, we started by writing a basic unit test. Literally: find a public method, call it with some arguments, and capture the return value through a debugger, then paste it into an assertion. Michael

Feathers calls this a Characterization Test [Feathers]. The point of these tests is to create an early detection system for breaking changes you might inadvertently introduce. We co-opted this technique to do some quick analysis about how isolated the class was and, in turn, how hard it would be to pull that class into a test harness. Write the test quickly and hastily, run it, and see what kind of runtime errors it produces. Those runtime errors help us focus on our goal of bringing this class under test and give us more visible targets for refactoring; we're down to the method level.

We found a common culprit. The method we were testing had a bunch of dependencies instantiated in its body. There was no hook in which to introduce a fake dependency we could control from within our test arrangement. We began to refactor the class to permit this behavior. Our goal was to be able to write specifications in the form of tests on this class, but we needed to begin by refactoring.

3.3 A Novel Practice Emerges

The refactorings were many and frequent. We introduced class instance variables, changed constructor parameters, and discussed principles such as Dependency Inversion, the strategy pattern, and test doubles. There was a delightful fusion of teaching, learning, and doing. Best of all, it was happening in everyone's day-to-day codebase. We weren't "doing TDD," but we were getting a lot of value out of understanding the qualities of a design that TDD tends to yield and learning how to use moves from TDD to better maneuver in legacy code.

So far, what I've described may seem familiar to you. We're using tests and refactoring to wrangle some messy code into something we can understand and perhaps even build on top of using a more TDD-like practice. What would happen next seemed like hive mind magic to me at the time.

Someone suggested we check back in with NDepend. How, exactly, had the metrics view changed since we last looked? NDepend has this nice feature in which it will baseline analysis runs and show you metrics on a time-series graph. Figure 3 gives you an NDA-friendly sense of what we saw after a few refactoring sessions.

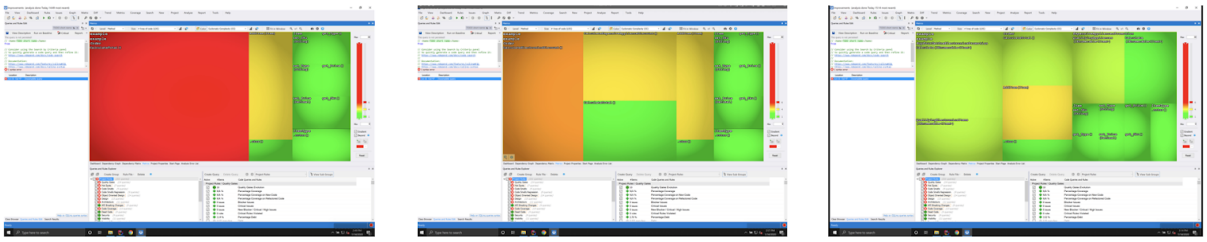

Figure 3 – Changes in code quality metrics over time across refactorings (right to left).

You can see how the code quality changes as our refactoring session progresses! We were able to perceive a difference in our SQALE maintainability index [SQALE], one measure of technical debt. Over several hours we were able to drive down the debt in a 300K+ line code base of about 1.2%. Participants in the room reflected that this telemetry gave enough evidence to go forward with our new practice of refactoring-first with the aid of metrics provided by NDepend. Furthermore, we uncovered a handful of refactoring combinations that would get code under test quickly. These, in turn, became reusable recipes that would be passed around and adapted internally.

I've since dubbed this "Refactoring with Telemetry" since it focuses on understanding design qualities by using code metrics to prompt discussion and create teachable moments throughout an extended refactoring session. I'm not trying to push this as the next big thing, a future best practice, but I use it frequently in my coaching and teaching approach. When the pattern of "how do we do this TDD stuff in this complex codebase" presents itself, this is my go-to technique for working toward the vision and value touted by TDD when working in legacy code.

4. ARCHITECTURAL MAPPING

Let's return to the architectural diagramming scenario I began this paper with. I suggested an ideal team would have a common understanding of their software architecture, to the point that they'd all produce similar diagrams when asked. This scenario derives from another instance where we discovered a new practice out of need and in a group learning setting.

4.1 Situational Awareness

This experience centers on a coaching engagement with a team at a large financial services organization in an immersive learning environment, a dojo. Part of the dojo involves doing discovery on a real business problem we'd like to solve during a team's 6-week challenge. Our standard discovery toolkit includes a practice called user story mapping [Patton]. Story maps help us visualize the path a user will take through our digital product as they attempt to complete a task that's important and valuable to them. Story mapping is an excellent technique for centering your backlog on the user. That's the context in which it works best.

We began story mapping with the group. As we took them through the steps, we encountered a lot of friction. We weren't getting the kind of engagement we usually do from groups. We weren't able to get into the flow of story mapping. One of the team members suggested the nature of their system was hampering our ability to apply the practice of story mapping.

We quickly learned that the system we were working on was a 99% back-end, API-driven, "lights out" kind of system, not uncommon in the financial world. The only user interaction with the system was a small dashboard that effectively acted as a monitoring window for automated business transactions and workflows. Story mapping, as part of a user-centered design workflow, did not apply.

4.2 Introducing an Engaging Experiment

The feedback of our failed attempt at story mapping yielded new insight that helped our coaching team guide the team onto a better method of discovery. I suggested we employ a mapping technique better suited to the terrain we now knew ourselves to be in: C4 Modeling [Brown].

C4 Modeling is an approach to architectural diagramming that turns your diagrams into zoomable maps (see figure 4).

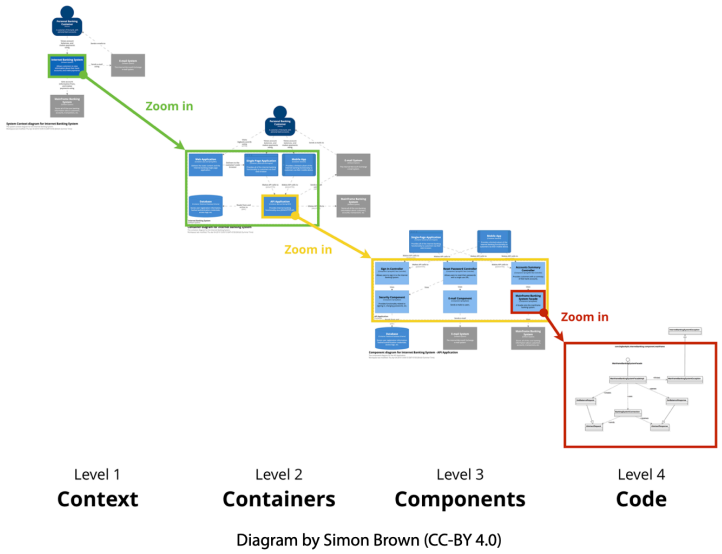

Figure 4 – C4 Architecture Diagrams & "Zoomable" Levels of Detail

At the highest level (context) we depict system context. This is the 10,000' view that lets us see which users, upstream and downstream internal systems, attached resources (shared databases, queues, streams, etc.) and third-party services comprise the ecosystem of dependencies in which our system lives.

From there, we can "pinch zoom" into containers, which tells us more about the distribution method and technical stacks of the pieces of our system. Zooming in on any container leads us to a deeper level, a component diagram, that gives us a sense of the logical arrangement of the system, the solution architecture. The lowest level takes us down to the actual code, showing us the structure and interaction between key modules, classes, or functions.

We tackled building a few layers of these maps, just enough to get into the problem we'd be working on, in a social manner using a lightweight version of mob programming. Each person on the team took a turn drawing diagrams as both team and coach took turns asking questions or sharing facts about the system (see Figure 5). The effect was very engaging and informative for the whole group.

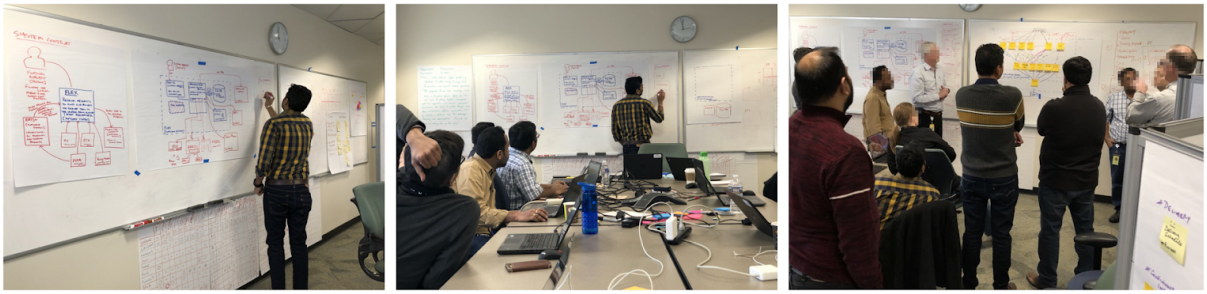

Figure 5 – Team Creating & Discussing C4 Diagrams

The point of discovery is for the team to learn about not only who you're developing for, your user, but also what system you plan to create or modify. The feedback we got from this session was great. Developers newer to the system emerged with much more awareness of what's going on in this large, potentially daunting codebase. The session went from sputtering to spectacular after we pivoted from story mapping.

4.3 Architecture as a Map, Features as Journeys

Stoked by insight and conversation, we moved on to create a plan to implement a new feature in the system. The team decided to convert their paper diagrams into a digital format and use this diagram to create a walkthrough of the changes necessary to implement the feature (see Figure 6).

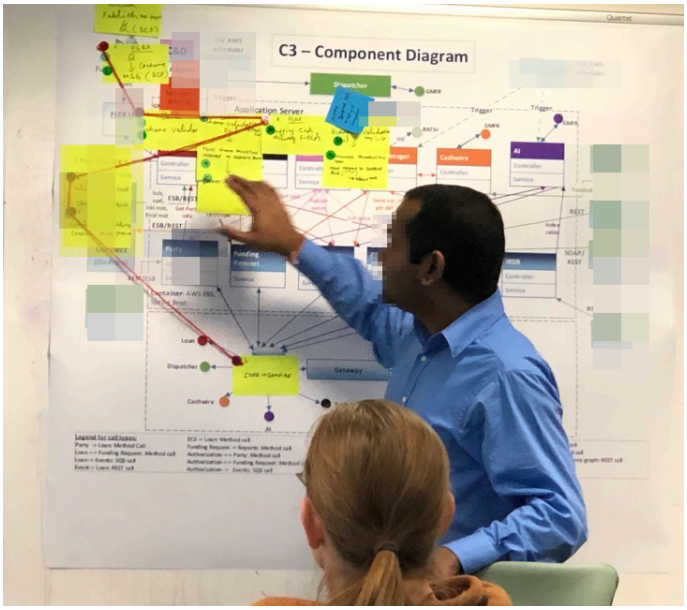

Figure 6 – Plotting a course with an architectural map.

At this point, we reached for a tool from story mapping itself and created journeys through the system. Unlike story mapping, this was not a particular user's journey but rather the flow of a piece of data originating from an upstream system and emerging as enriched messages consumed by a number of downstream applications. The map was different because the terrain was different, but once we had a map, we were able to unbundle the journeying technique and apply it to our new architectural map. The essential outcome was similar between our new, cobbled together practice and story mapping: team members can see a route overview of where their work is likely to take them.

5. TOWARD BETTER PRACTICES

Best practices often imply a specific context that makes them the best choice inside of a particular context. Rather than treat best practices as a hard goal, experiences like the above have convinced me we're better off starting with them as an aspirational vision, which we can refine as we learn our context over time. Doctrine is sometimes a useful starting point but becomes problematic when we insist on rigorous adherence at the expense of learning and situational awareness.

How might we place ourselves in a position where we can discover valuable, possibly novel practices? There are four components I focus on as I head into "adopt practice X" type engagements: separating content from context, creating a real-world learning environment, adopting an experimental mindset, and unbundling practice elements. I'll explore these components through the story of how we arrived at both Refactoring with Telemetry and Architectural Mapping.

5.1 Separate Content from Context

Best practices are content. Recognizing a fit between the current problem and previous experience is a matter of discovery and adaptation: discover context, adjusting to fit, repeat. Adaptation happens where content and context meet, and its outcome is potentially a new practice. Context discovery and practice adaptation isn't a mystical process; it requires that you listen and that a group is willing to try something new. This mechanism explains how we started with a goal of Story Mapping and arrived at Architectural Mapping.

Architectural Mapping came out of a desire to visualize a backlog outside of your typical work item tracking system. What we didn't know at the start is that the target system was devoid of user interaction. Story Mapping is a good fit when the terrain is user interaction.

Similarly, the fit between TDD and legacy systems can sometimes be awkward. I'm not saying it doesn't work, but you have to go about it differently. Often you have to write a few tests and tease code out of coupling before you get into the red-green-refactor loop.

5.2 Create a Real-world Learning Environment

Both practices emerged from a real-world learning environment. That is, we weren't learning through prepared lab exercises or simulation. We were confronting the team's actual problems and testing practices on their real system and desired impacts.

In the case of Refactoring with Telemetry, we had to get out of the controlled environment of a 2-day TDD class where context and content are carefully controlled variables. I much prefer a workshop setting over a classroom setting as it lets us explore practices and adaption to a team's situation simultaneously. Adaptation is a joint effort and not a puzzling exercise left to the student when they return to their day job.

5.3 Adopt an Experimental Mindset

When you're in such environments engendering a safe-to-fail ethos is paramount. Adopting an experimental mindset means that if an attempt doesn't go as planned, you process this failure as information, feeding that forward into your next effort. We are leaving the domain of authority and doctrine and entering the field of collaboration and discovery.

Creating these environments starts with setting the tone and often, for us consultants, with pre-negotiating our engagement model with the clients we're serving. As a coach, I help to nurture these environments by modeling behaviors consistent with an experimental mindset. Once the group sees me suggest a path, and if that path fails, they can see how I recover and suggest a pivot. Even better, they may suggest an alternative route themselves, thereby conscripting themselves into our friendly experimental conspiracy! Do this enough and you'll stop thinking of failure and start seeing productive feedback loops in play, seeing and exploiting them to the group's advantage.

5.4 Unbundle Practice Elements to Curate Novelty

So far, I've covered pattern matching and creating the conditions where original practices might emerge. But how do we construct and assemble new loops and workflows? If we were to crack open a given "best practice," we can find ideas for experiments to try on our way to finding value in the form of fit and adaptation.

Unbundling practices happened in both examples I've given. Refactoring with Telemetry is a new workflow that uses components of TDD. We're writing a few tests, and we're doing automated refactoring. Adding in a static analysis tool such as NDepend and finding a rhythm yields a new practice ideally suited to our context.

The same is true for Architectural Mapping. We pivoted from Story Mapping to C4 Model when it became clear that Story Mapping wasn't the right tool for our job. However, we did bring back a key component we

traditionally use in our story mapping practice: journeys. Instead of tracing a user's potential journey through the user interface of our software, we were previsualizing a message's journey through the components of our software.

A word of caution: unbundling isn't easy. It takes someone who knows the practice well and is ready and willing to tease components apart. Success requires some mastery: a willingness to break the rules after having worked within those constraints for some time. These components serve as rich experimental fodder, but only when we adopt the right mindset and create the conditions where experiments can happen.

5.5 Start with Learning, Experimentation, and Situational Awareness

Experiences like these have helped me reconcile some of the internal unease I've had with the term best practice. I now see best practices as a helpful jumping-off point for a new workflow that will serve a particular team. Best practices are harmful when we fail to take our context into account or are unwilling to bend the rules to match our situation. Without situational awareness, best practices can quickly become a quixotic time waster. When we combine this awareness with learning, experimentation, and flexible expertise, we're able to adapt best practice vision to a team's or organization's maximum benefit.

6. ACKNOWLEDGEMENTS

I'd like to thank the many teams that have demonstrated a willingness to go down sometimes weird and dead-end paths with me in search of adapting new practices. Unfortunately, non-disclosure agreements prohibit me from calling them out by name.

I'd also like to acknowledge Ed Tilford's role in helping us pivot to Architectural Mapping. Ed is a great coaching partner and a strategic thinker whose collaboration I value highly. Maneuvering a team onto value is always easier with him in the room.

Also, a huge thank you to my shepherd in the XP 2020 Experience Report writing process, Dr. Cherifa Mansoura. Without her economical and well-placed words of wisdom and guidance, I'd still be writing this paper.

REFERENCES

[Brown] Simon Brown, 2020. *The C4 Model for Visualizing Software Architecture: Context, Containers, Components, Code*. Available online at https://c4model.com/. [Accessed 20 May 2020]

[Feathers] Michael Feathers, 2004. Working Effectively With Legacy Code. Prentice Hall.

[Fowler] Martin Fowler, 2018. Refactoring: Improving the Design of Existing Code (2nd Edition). Addison Wesley Professional.

[NDepend] NDepend, v2020.1.1, 2020. ZEN PROGRAM LTD.

[Patton] Jeff Patton, 2014. User Story Mapping: Discover the Whole Story, Build the Right Product. O'Reilly Media.

[Snowden] David J. Snowden, 2007. *Harvard Business Review: A Leader's Framework for Decision Making*. Available online at https://hbr.org/2007/11/a-leaders-framework-for-decision-making. [Accessed 20 May 2020]

[SQALE] Jean-Louis Letouzey, 2016. *The SQALE Method for Managing Technical Debt*. Available online at http://www.sqale.org/wp-content/uploads/2016/08/SQALE-Method-EN-V1-1.pdf [Accessed 20 May 2020]