Integrating Lean Processes and Engineering Discipline into Work Culture Over 20 Years: An Experience Report

Doug Durham
Don't Panic Labs
Lincoln, Nebraska, USA
Email: ddurham@dontpaniclabs.com

Bonita Sharif
University of Nebraska - Lincoln
Lincoln, Nebraska, USA
Email: bsharif@unl.edu

***Abstract*—The paper presents an experience report for Don't Panic Labs (DPL), a software and consulting company, outlining the cultural shift in building software using lean processes and engineering principles over 20 years. The shift was intentional due to failures of prior projects. Challenges are identified along with opportunities to address them. An evolution of the company culture is explained via themes that were incorporated chronologically over time, which resulted in reducing errors in judgment, leading to reliable and predictable successes. Two factors, namely, the percentage of rework and co-creation sessions for efficient iteration of requirements, stand out as significantly impacting project success. Lessons learned with discussion and impact to the profession and education are presented.**

***Index Terms*—experience report, lean processes, agile, systems**

I. Introduction and Background

Much has changed in the world of software development over the last 30 years [1], [2]. From the focus on client-server and desktop application development in the 1990s, to the advent of distributed computing in the early 2000s, followed by the rapid adoption of mobile devices as a computing client in the early 2010s. In 2011, Marc Andreessen (creator of Mosaic, the first Web browser) wrote an essay [3] discussing how software was *eating the world* and that some day there would be no industry left untouched by software. The last 10 years have seen an explosion in cloud-based systems and services that enable sophisticated service-based systems. The advancement of these technologies has allowed businesses to see software as a means of solving difficult problems and to enable them to differentiate themselves from their competition. Recently, the maturity and widespread availability of machine learning and other AI-based systems [4], [5], [6] and services have opened up a whole new set of possible problems to solve.

Along the way, the development community has been absorbing and adopting processes and methods based upon the principles of the Agile Manifesto [7] with the hope that these will help them succeed with their software projects. Sadly, as an industry, we still struggle to consistently create successful outcomes (budget, quality, timeline, user satisfaction). Stakeholders do not have the same confidence level in our ability to manage these projects as stakeholders of other engineering disciplines. Our industry's inability to effectively manage the complexity of developing these complex systems results in a significant percentage of projects being over budget or canceled outright [8]. Even projects that get delivered often result in significant effort devoted to rework [9] (time spent on defects) during the support and maintenance phase of the life-cycle, draining resources away from other priorities.

Systems engineering processes and methods at McDonnell Douglas' MacAir division (where the first author initially worked), that built software for military aircraft avionics systems integration, in the early 1990's were rigorous and disciplined, if not heavy and burdensome. The net result was a significant shared understanding of what needed to be done, what the status of that effort was, and the quality of the efforts at the end. This shared understanding spanned not only various stakeholders within the company but also the customer and end-users. While timelines often changed, it was extremely rare for a project to fail due to technical, quality, or usability reasons. By contrast, the normal experience working in a commercial software company that builds software for health-care practice management and electronic commerce from the mid-1990's until around 2005 was anything but rigorous and disciplined [10]. This experience of dealing with different domains and companies can only be described as **chaos**.

Relying solely on the principles of the Agile Manifesto [7] will not allow us to manage the ever-increasing complexity of the software systems being developed today. The lack of widespread technical maturity of our development community and their lack of understanding of the software engineering body of knowledge (SWEBOK) [11] will result in significant errors in judgment, lost productivity [12], frustrated stake-holders, and unpredictable outcomes. The principal hypothesis is that software development could achieve the predictable outcomes of other engineering disciplines, without sacrificing the agility that comes with building systems in software and without requiring exceptional efforts or talent from the teams involved [13]. The journey and evolution (Section III) is the result of efforts to prove this hypothesis.

This paper will review the experience of Don't Panic Labs (DPL), which recognized the multiple dimensions of complex-

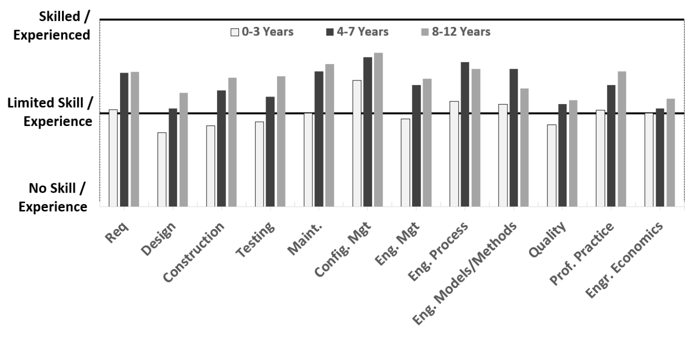

Fig. 1. Survey results: Knowledge of SWEBOK [11] among 73 practitioners.

ity to be managed and that individuals make decisions daily that contribute to the success or failure of these projects. This realization led us to take active steps to go beyond simple agile methods to identify and integrate tools, techniques, methods, and processes that have enabled us to effectively manage these dimensions of complexity to create more predictable outcomes by reducing errors in judgment resulting from leaving those judgments and individual decisions to chance.

II. Challenges and Opportunities

According to the CHAOS Report [8] from The Standish Group, in 2020 only 31% of all software projects are on time and on budget. There are still a significant number (19%) of projects failing. These statistics are for projects of all sizes. A corollary issue is the impact of these struggles on the productivity and efficiency of the development teams, which comes in the form of rework. *Rework* [9] is another term for corrective maintenance and is the amount of time spent addressing defects in a project after it was claimed complete and correct, regardless of the source of the defect. It is estimated that software projects spend more than 50% of their effort on rework, leaving only half of the throughput of an organization available to advance the software with new features [9], [14]. From our observations and experience having worked on numerous projects, we have identified two causes for this inability to have reliable and predictable success.

Lack of Software Engineering and Design Literacy: Software Engineering, as a field of undergraduate study, is still uncommon. There are a number of experienced individuals in industry who entered the field with little, if any, formal college education. In addition, we have a large percentage of programmers that identify as end-user programmers [15]. Given the diverse backgrounds and education of individuals, there is a broad lack of understanding and familiarity with the software engineering body of knowledge. We surveyed 73 practitioners at two local dev conferences from varied organizations with varying levels of experience and it is quite clear that there is a basic lack of understanding of the various knowledge areas (see Figure 1). While practitioners understandably gain some skill and experience on-the-job, it is clear from Figure 1 that they are not starting with a sufficient baseline. Without a foundational understanding of this body of knowledge and with only one's experience and informal self-education to lean on, it is not hard to understand that errors in judgment will often occur. This ultimately results in the struggle many teams face in delivering software designs and implementations without simultaneously introducing defects [16] and code smells [17].

Inefficient Iteration and Learning: The second cause we have observed is related to how development teams iterate and learn throughout the development process – both in terms of efficiency and effectiveness. One need only look at the amount of effort development teams spend on rework to understand that 1) there is a lot of iteration and learning happening during the coding phase and 2) there are likely opportunities to improve efficiency in how we are learning and iterating [9]. One of the key observations we made is that it is challenging for teams to ensure they have a shared understanding without deliberate effort. There seems to be a bias in many organizations to limit the amount of critical thought and analysis upfront in favor of getting in and writing code. While the latter does not mean a project will ultimately fail, it should be acknowledged that iterating on the design of a software system in code is way less efficient than using critical thought and abstractions (wireframes, white papers, architecture design documents, etc) before implementation starts. Missed milestones and budget overruns are likely if iteration is not done correctly. One of the key benefits we find with structured tools that require individuals to express, visually or in words, their understanding of the design or the requirements is that it often reveals 1) differing understandings between team members of details that were thought to be well understood, and 2) implicit assumptions people make about a system that had not previously surfaced.

Opportunities: We fundamentally believe it is not unreasonable to expect project success (on time and on budget) in the vast majority of software projects. The key to obtaining this level of success hinges on our ability to move the software development industry in close alignment with the other engineering disciplines. It is generally accepted in our industry that a "rock star" programmer can be 10 times more productive than an average developer. In order for predictable outcomes to be the norm in every organization, we need to eliminate our reliance on "rock star" performers as it is not sustainable. The path away from this reliance is to elevate the rest of the development community by 1) educating them on the SWEBOK, 2) standardizing a set of system design patterns that can be understood and effectively put into practice by the average developer, and 3) create a series of processes and tools that enable critical thinking and shared understanding in all phases of development. The end result of this should be complex problems that are routinely and predictably solved by "rank and file" engineers. This opportunity for training also works well if tied to the local university curriculum (e.g., [18]) where software engineering degrees are offered, which is what DPL supports, engages in, and promotes.

III. Evolution of Company Culture

Don't Panic Lab's culture and processes have evolved over roughly the last 20 years. The primary triggers are continuous reflection and identification of recurring challenges and inefficiencies that were preventing predictable/successful outcomes.

A. *Formalizing Management of Projects*

Coordination between stakeholders and the development team was done informally and put a lot of pressure on developers. A need for a project manager role, who would be accountable for coordinating between all stakeholders and the developers, to ensure proper tracking and prioritization of requirements was created. This role formalization led to additional benefits including the development of a method for tracking our "system" for developing software along with the value propositions [19] of each component of this system, with periodic reviews to ensure we were only following processes that created the value we desired. Rather than simply taking a feature or other requirement and applying some ad hoc estimate, we defined a set of possible estimates that include half day, 1 day, half week, 1 week, or more than a week. If a requirement is determined to take more than a week, it is further decomposed such that no individual estimate is greater than a week and, ideally, all individual estimates are half week or less. The basic premise is that large individual estimates represent significant uncertainty in understanding a given requirement and by breaking the requirement down reduced the risk of not hitting our estimates.

B. *Structuring Quality Assurance Efforts*

A recurring challenge was knowing whether our quality efforts were effective and when the software was ready to be released into production. It was also becoming very difficult to support the complex systems in production in terms of debugging issues that arose and having the confidence to refactor or redesign portions of the systems with no unintended changes to behaviors. Addressing these concerns involved three separate efforts. *First*, the quality assurance processes were changed to ensure that there would be formal test plans developed for testing new functionality and regression testing existing functionality. These formal test plans made it possible to track progress of test plan execution and success, making it easier to determine release readiness and project release timelines. The *second* effort was to adopt a formal process for collecting additional data on defects that allowed for the calculation of a defect detection rate (percentage of defects found before the end user found them) with the goal of keeping it above 90%. The *third* effort was the adoption of automated testing tools for unit, integration, and system-level testing. Not only did this improve the quality of the code in development, but it also created a suite of automated regression tests, which reduced the risk of addressing production issues with hotfixes.

C. *Shifting Quality Assurance Accountability to Developers*

Even with the structuring of quality assurance efforts, we still were experiencing frequent *thrashing* of code changes where 1) a developer would make code changes, 2) release code to test environments, 3) quality assurance would quickly find a defect, 4) bug would be written, 5) developer would make code changes... repeating the cycle many times. We came to the realization that the quality assurance team became a "crutch" for the development team, who felt no real ownership or accountability for making sure their code was solid and defect free before releasing to the test systems. It was felt to be critically important to shift this accountability to the developers and the solution was to reduce, or completely eliminate, the availability of quality assurance personnel on future projects. This rapidly changed the culture of the development team and quickly shifted the focus to increased investment in test automation and adoption of continuous integration tools that gave the development team immediate feedback on the quality of the code being merged into primary branches. When the continuous integration quality checks failed, there was an immediate effort to resolve the issue before moving forward with other development efforts. It also triggered the adoption of a layered approach to quality where no one quality practice (unit testing, desk checking, integration testing) could achieve our defect detection goals but the combination of all of these practices during a project would get us there.

D. *Increasing Developer Efficiency*

We became aware of an essay written by Paul Graham of Y Combinator [20] that profoundly changed the way we thought about the impact of daily interruptions and how we viewed barriers to developer productivity. The basic premise of the essay is that people who are "makers" (i.e., developers) need large blocks of time to get into the problem before they start to really become productive. Interruptions every hour or two for a meeting kept this from happening. *First*, we immediately scrutinized both the meetings that we were requiring developers to be a part of and when those meetings were occurring. As a goal, we attempted to have these meetings early in the day. The *second* significant change involved how we configured the development environments of the individual developers. Historically, testing the code we were working on would often involve interaction with one or more development servers that would be *shared* by all members of the team. Shared database servers were the worst offenders. Solving this problem involved two significant changes to our development environments: 1) we choose technologies and system architectures that allowed developers to do most, if not all, integration testing on their own machines, independent of other developers, and 2) we adopted a configuration management strategy and toolset (e.g., DbUp)[1] for database schema configurations that allowed developers to make changes on their local machine database that would be merged into a common database configuration repository.

E. *Increasing Design Stamina*

The realization we had is that there was no coherent design process enabling the system to be built as if one person had

[1]https://dbup.readthedocs.io/en/latest/

done all the design. Instead, we had systems that were built with each part of the system reflecting independent design decisions (design whim) of the person writing the code. Compounding this problem is that the code and design reflected the skill (or lack thereof) of the developer even though there were highly skilled engineers who could improve the design. The changes we made to address these problems have likely had the most significant impact on our development maturity and our ability to design and develop systems with confidence. Addressing the lack of coherent system design required us to adopt a methodology for decomposing systems that could be used by experienced architects and followed by development teams. We were introduced to such a methodology that was developed by Juval Löwy[2] that was inspired by the work of Parnas who strongly advocated for designing systems for change using a technique for decomposition that he called information hiding [21]. Two key processes were adopted: a structured design analysis document template to reason through requirements and a code review pull request model. This impacts developer efficiency by allowing them to move between teams with little technical friction.

F. Interaction Design Specialization

Until very recently (last 10-15 years), it has been possible to design user interfaces for software systems by allowing individual developers to make layout and workflow choices or to use layout automation tools and templates for web pages. The increasing sophistication of more contemporary software systems, along with the need to support multiple platforms (e.g., multiple web browsers and computing platforms such as desktop, laptop, tablets and mobile devices) has made this approach not only impractical but also a liability that can impact the success or failure of a system. Recognizing this threat, we began to hire and develop specialized talent that was focused on developing an understanding of best practices for interaction design and the process and techniques that can be used to rapidly express user experiences in a way that validates the design with the end user and stakeholders and provides sufficient guidance to the developers to ensure implementation of the intended design. Separating the responsibility for designing the user experience from the implementation requires close coordination of both parties to ensure that what is designed is feasible and to limit the use of non-standard custom controls in favor of existing reusable controls.

G. Increasing Visibility into Project Progress

Even with all of the above changes to reduce uncertainty of outcomes and errors in judgment, there is always going to be a remaining risk that can impact the outcome (budget/schedule) of a project. Identifying the presence of this risk as early as possible can provide more options for mitigation as opposed to when the risk is identified late in development. One of the key challenges preventing early risk identification is the ability to reliably and confidently understand project progress at any point in time. Addressing this challenge involved adopting established *earned value management* [22] as a technique for tracking project progress. Leveraging our previously mentioned discrete estimation technique (Section III-A), we are able to establish an anticipated project schedule based on the estimates of planned work and then track that against the schedule for both completed work and total actual effort. Adopting this tool has led to earlier identification of technical risk and requirements uncertainty and has become a valuable tool for collaborating with stakeholders to manage constraints.

H. Increasing Iteration/Learning before Coding

Learning and iterating based upon the learning is necessary for all projects to increase the likelihood of a successful outcome relative to customer satisfaction. Historically, the majority of this learning and iteration occurs within the coding phase, but it is expensive if done this way. We strongly believed that moving to an approach where, 80% of the learning and iteration happened prior to coding would yield great benefits to our clients and project outcomes. Rather than relying on product owners and stakeholders sole understanding of the requirements, a lightweight, structured co-creation process is used that allows for the discovery of the impacts and outcomes that are motivating the project. When possible, these interaction design artifacts are augmented by developing testable requirements (e.g., using Gherkin Syntax[3]) which allows stakeholders and developers to collaborate in the creation of human-readable, non-technical test cases that uncover edge cases and alternate outcomes prior to development and also provide a strong basis for acceptance testing criteria.

IV. Lessons Learned

Our own personal journey was a series of steps in which we tackled some aspects of our inefficient development processes. When we were asked by other organizations to help with their own transformation, it was tempting to try to *turn the entire ship* via a rapid set of simultaneous changes across the organization. Our instincts have told us this is not feasible, especially when you also have to bring along an entire team of motivated but understandably skeptical people. We have found it much more productive and sustainable to change the organization via *evolution*, as opposed to *revolution* and to prioritize the changes to meet the organizations *where they are at* to give them confidence through incremental successes.

Another important aspect of the process of this type of change is understanding the importance of trusted leadership that is leading the change. Having someone "in charge" who can build consensus within the team on what is to be changed and why it is important will help get everyone on board. This strong leadership is also important when consensus cannot be reached and command decisions are necessary and in the best interest of the organization. At some point during our own evolution, we began to realize that the real problem we were trying to solve was to eliminate outcomes being *left to chance*. Once we identified this core goal, it became easier to identify *where the holes were* in our processes and methods. We could

[2]https://www.idesign.net/Services/System-Design

[3]https://cucumber.io/docs/gherkin/reference/

also begin to prioritize the desired changes based on the risk associated with the *hole* that was found.

Finally, once we started seeing a significant change in the successful outcomes of projects and the long-term maintainability of our system designs, we noticed that the rank and file members of our team moved from adoption of our patterns and practices with successful outcomes, to strong advocacy for them. Once we got the heavy flywheel spinning, the momentum continued carrying us forward, and more and more members of the team were focused on ensuring we were following our processes and looking for ways to improve them. This is how we knew we sustainably changed the culture.

We briefly elaborate on how the evolutionary steps (Section III) are measurable. For *project management formalization*, improvement in earned value performance vs. plan (project timeline and budget adherence (+/- 25%)), and stakeholder satisfaction (via the net promoter score [23]) need to be tracked. For *quality assurance* efforts, project-level defect detection rate (within a sliding window) is a key metric. DPL targets a 90% defect detection rate. A good metric for *developer efficiency* would be percentage of time spent on rework [12] from the last 6-12 months. *Design stamina* can be measured by change in velocity over time, as new features are added. One can track if something that was easy to change five years ago is now more difficult, which means increased friction. Good estimation plays a role as well. Developers can be prompted as part of a pull request to comment on whether the initial estimate was good. The rest of the evolutionary steps point to process and role changes. For example, we currently have invested in five dedicated interaction designers at DPL. They work with end users to create workflows as part of the design and requirements process during iteration and co-creation sessions. As part of employee engagement, DPL has used Lattice (https://lattice.com/) for the past 4 years, which produces a pulse score for managers each week, because team dynamics / human factors [24] are also critical.

V. Discussion and Impact

Our experience evolving our organization to reduce errors in judgment and to build and adopt a comprehensive "system" for developing software has allowed us to achieve regular success on dozens of innovation projects over the last 10 years. As an example, Figure 2 demonstrates how adopting many of the themes above by one particular project substantially reduced the amount of rework experienced, cutting it almost in half within a 6-year period. We have noticed a strong desire within our development teams to protect this culture and to actively avoid opportunities with other organizations where we might have to subject ourselves to less rigorous and disciplined patterns and practices. We have come to realize that the success our development teams experience aligns with a lot of the motivations for entering this career field. By and large, we have been able to eliminate those negative experiences (missed milestones, poor quality, frustrated end users, etc) that many development teams still experience.

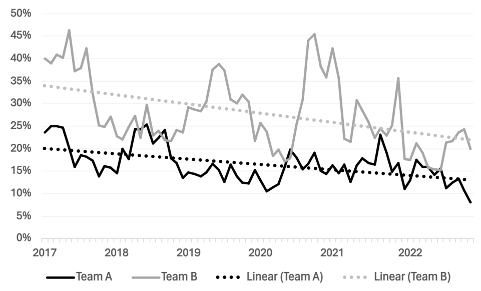

Fig. 2. Rework (labor hours spent on defects/labor hours spent overall) in two teams (named Team A and Team B for anonymity) between 2017-2022. Work items were coded as rework by project managers at DPL.

Another impact of this work is on the approach to education [25] and professional development. Software development, like many other fields of engineering, involves a number of tradeoffs when making decisions along with non-deterministic solutions to system designs. When we look at the way most education is structured, it tends to be focused on more deterministic, small-scale problems. As a result, most people coming out of education programs lack 1) experience with the non-deterministic nature of the design of larger, more complex systems, 2) sufficient knowledge or experience with core principles such as cohesion and coupling [26] and information hiding such that they can use these principles to make good judgments within design decisions, and 3) very little familiarity with the vast majority of the knowledge areas within the SWEBOK (evidenced by our survey in Figure 1). The net result is that we often see organizations where there is little understanding of the impact of their decisions and a lack of appreciation for the need to manage the complexity of their projects to ensure positive outcomes. Their solution to meeting project challenges is just more coding ("fingers on the keyboard") as opposed to more critical thinking and leaning on what is known about our field. We strongly advocate that the approach to educating future software engineers must change to address these deficiencies. Some of these changes can be done by incorporating co-creation sessions and industry working sessions like DPL has provided at University of Nebraska - Lincoln in the software engineering requirements elicitation course and senior capstone projects with faculty.

VI. Conclusions and Future Work

We present an overview of the significant challenges Don't Panic Labs software development teams faced that led to errors in judgment and unpredictable outcomes of their development projects. We also review steps the company took to evolve the organization and culture to successfully address these challenges over a 20-year period. As part of future work, we plan to conduct a field study tracking the evolutionary metrics such as work items, estimation, and rework on future projects. Furthermore, we seek to create educational material/case studies that would allow other organizations and universities to benefit from these experiences and confidently and effectively evolve their own organizations and software engineering teaching practices with similar success.

References

[1] T. D. LaToza, "Crowdsourcing in software engineering: Models, motivations, and challenges," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2019, pp. 301–301. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE-SEIP.2019.00043

[2] F. P. Lima, W. Viana, M. F. Maia, R. C. Andrade, M. Castro, J. F., and L. S. Rocha, "Ubiquitous software engineering: Achievements, challenges and beyond," in *2013 27th Brazilian Symposium on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2011, pp. 132–137. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SBES.2011.33

[3] Marc Andreessen, "Why software is eating the world," https://a16z.com/why-software-is-eating-the-world/, 2011, accessed: 2024-08-15.

[4] C. K. Tantithamthavorn and J. Jiarpakdee, "Explainable ai for software engineering," in *Proc. of the 36th IEEE/ACM Intl Conf on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, p. 1–2. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678580

[5] S. Martínez-Fernández, J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. M. Vollmer, and S. Wagner, "Software engineering for ai-based systems: A survey," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, apr 2022. [Online]. Available: https://doi.org/10.1145/3487043

[6] A. Bendimerad, Y. Remil, R. Mathonat, and M. Kaytoue, "On-premise aiops infrastructure for a software editor sme: An experience report," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1820–1831. [Online]. Available: https://doi.org/10.1145/3611643.3613876

[7] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2001. [Online]. Available: http://www.agilemanifesto.org/

[8] The Standish Group International, "The chaos report," www.standishgroup.com/sample_research/PDFpages/Chaos1994.pdf, 1994.

[9] S. Tockey, *How to engineer software: a model-based approach*. Hoboken: John Wiley IEEE press, 2019.

[10] J. Voas, "A baker's dozen: 13 software engineering challenges," *IT Professional*, vol. 9, no. 2, pp. 48–53, 2007.

[11] P. Bourque and R. E. Fairley, Eds., *SWEBOK: Guide to the Software Engineering Body of Knowledge*, version 3.0 ed. Los Alamitos, CA: IEEE Computer Society, 2014, http://www.swebok.org/.

[12] C. Jones, *Applied software measurement: global analysis of productivity and quality*, 3rd ed. New York, NY: McGraw-Hill, 2008.

[13] D. Durham and C. Michel, *Lean software systems engineering for developers: managing requirements, complexity, teams, and change like a champ*. New York: Apress, 2021.

[14] D. A. Wheeler, B. Brykczynski, and R. N. Meeson, *Software Inspection: An Industry Best Practice for Defect Detection and Removal*, 1st ed. Washington, DC, USA: IEEE Computer Society Press, 1996.

[15] M. M. Burnett, "End-user software engineering and why it matters," *J. Organ. End User Comput.*, vol. 22, no. 1, pp. 1–22, 2010. [Online]. Available: https://doi.org/10.4018/joeuc.2010101904

[16] B. Eken, "Assessing personalized software defect predictors," in *IEEE/ACM 40th Intl Conf on Soft Eng: Companion Proceedings (ICSE-Companion)*. Los Alamitos, CA, USA: IEEE CS, jun 2018, pp. 488–491. [Online]. Available: https://doi.ieeecomputersociety.org/

[17] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[18] M. M. Alhammad and A. M. Moreno, "Integrating user experience into agile: an experience report on lean ux and scrum," in *Proceedings of the ACM/IEEE 44th Intl. Conf. on Software Engineering: Software Engineering Education and Training*, ser. ICSE-SEET '22. New York, NY, USA: ACM, 2022, p. 146–157. [Online]. Available: https://doi.org/10.1145/3510456.3514156

[19] R. Bavani, "Global software engineering: Challenges in customer value creation," in *5th IEEE Intl Conf Global Software Engineering (ICGSE 2010)*. Los Alamitos, CA, USA: IEEE CS, 2010, pp. 119–122. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICGSE.2010.21

[20] P. Graham, "Maker's Schedule, Manager's Schedule," https://www.paulgraham.com/makersschedule.html, accessed: 2024-02-08.

[21] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, p. 1053–1058, dec 1972. [Online]. Available: https://doi.org/10.1145/361598.361623

[22] P. M. Institute, *The standard for earned value management*. Newtown Square, Pennsylvania, USA: Project Management Institute, Inc, 2019.

[23] J. G. Dawes, "The net promoter score: What should managers know?" *Intl Journal of Market Research*, vol. 66, no. 2-3, pp. 182–198, 2024. [Online]. Available: https://doi.org/10.1177/14707853231195003

[24] P. Lenberg, R. Feldt, and L. Wallgren, "Human factors related challenges in software engineering – an industrial perspective," in *2015 IEEE/ACM 8th Intl Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2015, pp. 43–49. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CHASE.2015.13

[25] C. Ghezzi and D. Mandrioli, "The challenges of software engineering education," in *27th International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, may 2005, pp. 637–638. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE.2005.1553624

[26] S. Tiwari and S. S. Rathore, "Coupling and cohesion metrics for object-oriented software: A systematic mapping study," in *Proceedings of the 11th Innovations in Software Engineering Conference*, ser. ISEC '18. New York, NY, USA: ACM, 2018. [Online]. Available: https://doi.org/10.1145/3172871.3172878