Proceedings of the Design Society, Volume 5: ICED25
https://doi.org/10.1017/pds.2025.10095

Bridging theory and practice in software design management: insights from practitioners

Julie Johnson✉, Ada Hurst and Frank Safayeni

University of Waterloo, Canada

✉ julie.johnson@uwaterloo.ca

ABSTRACT: Research into the foundational theories and management of software design remains limited. A 2010 workshop was convened to explore professional software development practices. The workshop sought to foster collaboration between the software engineering and design communities by examining foundational aspects of software design. Building on that workshop's objectives, this paper investigates contemporary professional software design practice and its management within organizational contexts. It is informed by findings from three interviews with experienced software design managers. This work addresses an important gap by examining software design management through the lens of design rather than solely from a project management perspective. Additionally, it contributes to the development of a general theory of software design by integrating diverse theoretical frameworks.

KEYWORDS: design management, software design, design practice, design process, organizational processes

1. Introduction

"Software entities are more complex for their size than perhaps any other human construct" (Brooks, 1987).

Over time, software is becoming increasingly complex (Cardoza, 2020). While software design shares elements with other types of engineering design, it is acknowledged as having a "*distinctive character*" (Petre et al., 2010, p. 535). Software is built by teams and in organizations, yet organizational influences on software design are not often studied (Whitworth & Biddle, 2007). Literature on software development management is sparse (Kalliamvakou et al., 2019), mostly focused on software project management rather than on design management (e.g., Shastri et al. (2017)). To manage a design team is to focus both on current design outputs and on future design capabilities of the team. What is the desired trajectory for such a team? What does "maturity" look like for modern software development teams? Literature on the underlying theories of software engineering is similarly sparse. There is extensive literature on methods, but not necessarily on the theories underlying them (Johnson et al., 2012). Theory development in the area of software development has been described as "*slow to develop*" (Hall & Rapanotti, 2017). There is a "*patchwork*" of theories (Wieringa, 2014) such as the works of Davis (1995) and Endres and Rombach (2003), but no "*substantial theory of software engineering*" (Hall & Rapanotti, 2017). Design researchers have similarly acknowledged that software design is "*insufficiently studied*" (Petre et al., 2010). How can you effectively manage a task that is not well understood?

In 2010, an NSF-sponsored workshop - Studying Professional Software Design 2010 (SPSD 2010) - was held with the objective of uniting the design and software engineering communities in the study of software design (Petre et al., 2010). A research agenda was presented, suggesting that future research focus on:

- Studying design throughout the software process, not just early conceptual design

- Studying the different "*modes of working*", specifically software design as done by individuals as compared to software design done in large, distributed teams
- Studying "*different roles and expertise*", acknowledging that people with vastly different amounts of experience participate in software design.

However, since that time, there has been relatively little progress in the study of software design within the design community. The management of software design is a particularly challenging area of research because it requires knowledge of many different areas including design in general, software design in particular, and management and organizational theory. Building upon the aims of the 2010 workshop, this paper aims to advance the understanding of modern professional software design practice and its management within modern organizational settings. It draws upon the findings of three interviews with practicing software design managers, part of a larger study on modern software design practice, and offers some preliminary reflections on the extent to which prominent software design and software design management theories adequately model modern software design practice. The rest of the paper is structured as follows. Section 2 provides a brief literature review on software design theories and software design management. Section 3 describes the method for the interview study. Results from the interviews are provided in section 4, and findings and contributions are highlighted in Sections 5 and 6.

2. Background

2.1. Software design theory

Petre et al. (2010, p. 535) state that there is "*no single, agreed-upon way to consider the software process in the software engineering research community*." This community does not have consensus around its core theories and instead has various "*collections of propositions*" (Johnson et al., 2012). For example, Davis (1995) described 201 "principles" under the categories of requirements, design, coding, testing and management. Many of these are reflected in the later Agile Manifesto (Beck & Al., 2001). Similarly, Endres and Rombach (2003) describe 50 "laws", 25 "hypotheses" and 12 "conjectures". Like Davis', their work covers a breadth of topics including requirements, design, construction, testing, maintenance, and management. These have been described as examples of "*middle-range*" (e.g., non-universal) software development theories (Wieringa, 2014) - usable, context-dependent theories.

In the 1970s, the "waterfall" model of software design was introduced, to bring order to the process of designing and implementing software (Brooks, 2010). Waterfall projects assumed a fixed scope that was delivered only at the end of the project. Upfront planning and documentation were heavy, as was change management. Many projects failed, because feature needs changed during the duration of the development. In response, the Agile Manifesto (Beck & Al., 2001) was created in 2001. The Manifesto is a practitioner-derived set of 4 values and 12 principles intended to address issues seen in projects. It promotes "early and continuous delivery" of working software, collaboration with the customer, welcoming changing requirements, and building motivated, "self-organizing" and highly collaborative teams. Arguably the Agile Manifesto may be viewed as a practitioner-derived middle-range theory.

A common focus of many software design theories is determining key activities and phases in the process. An early attempt by Brooks (1987) divides software design into two types of tasks: *essential*, which he characterized as defining the conceptual structures of the software and *accidental*, which he defined as the representation or implementation of those conceptual structures in programming languages. He hypothesized that historical productivity gains were related to making accidental tasks easier. He believed that the challenging part of software development was the "*specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation*" (p.11). As such, he thought that building software would always be difficult.

More recently, Highsmith and Orr (2013) described software development as a complex adaptive system where information exchange is the means of interaction. Their model of software development includes three iterative activities: speculate, collaborate, and learn. "*Speculate*" means to "*postulate a general idea of where we are going, and put mechanisms in place to adapt—to explore the territory*" (Highsmith & Orr, 2013). "*Collaborate*" refers to shared creation and discovery. "*Learn*" refers to expertise gained through experience. Hall and Rapanotti's (2017) model comprises three phases: framing (e.g., a need which exists in an environment), representing (e.g., the characterization of problems), and transforming (e.g., changing the physical world to meet a need). They also introduced a "*three ellipse*" model based on

humans, the environment and technology. Their theory describes the interfaces between the domains and suggests prioritizing the social subsystem (something that often doesn't happen in practice).
Kannengiesser and Gero (2015) used the Function-Behaviour-Structure (FBS) ontology of design to analyse and compare engineering, software and service design. They hypothesized that design is independent of domain. Their simulation models did find commonalities, particularly that function and behaviour issues occur at the start of projects and structure and structure-behaviour issues occur later. It may be notable that their models used the Rational Unified Process (RUP), due to its detailed description (RUP had its peak popularity in the 1990s (Ambler, 2023)). Earlier, Gero and Kannengiesser (2007) described a model of "emergent" design. In that model, the waterfall model of software design is mapped onto the steps of the FBS framework. Additionally, three types of reformulations are identified: unexpected changes in requirements (e.g., changes to function and behaviour) and unexpected changes to the design (e.g., changes to structure). Interestingly, in a more recent examination of software design using surveys and case studies, Ralph (2016) found no evidence of an "*FBS subculture*" in software design (e.g., no separate analysis, design, coding, or testing phases) and argued that software development is better modelled as an "*ad-hoc oscillation between making sense of the project context (Sensemaking) simultaneously improving mental representations of the context and design space (Coevolution) and constructing, debugging and deploying software artifacts (Implementation)" (p.232)*.
Finally, Adolph and Kruchten (2011) report that their interviewees did not distinguish between requirements gathering, planning, and design phases; rather, "*all were described as negotiation*" (p.52). The authors propose a process of "*reconciling perspectives*" that includes the activities of reaching out, negotiating consensus, bunkering (software development), and accepting (Adolph & Kruchten, 2011). In summary, we see various authors trying to distil key activities and characteristics of software design. We see commonalities but also different (and sometimes contradictory) lenses or approaches.

2.2. Software design management

Our review of the literature on software design management found few relevant publications, and fewer still published in the last five years. In addition, although software design appears to have a unique character compared to other types of design, searches for comparisons between software design managers and other kinds of engineering managers did not produce any meaningful matches. The dearth of research is even greater around software middle management. There are a handful of books related to middle management, but none specifically focused on software development.
The few studies that do exist focus on the characteristics of successful software engineering management. For example, Kalliamvakou et al. (2019) interviewed engineers and managers at Microsoft and found successful managers *were technical, enabled autonomy, built relationships and team culture, and guiding and inspired their team*. Their findings were consistent with Garvin's (2013) at Google. Evans (2023) asserts that a software manager's role consists of four main activities: engineering (e.g., design), managing (e.g., communications, managing risk), transitioning (e.g., growing the team, handing priorities), and big picture thinking (e.g., team design, and retention).
Search results for software management often relate to software project management rather than people or design management. Shastri et al. (2021) found that, in addition to traditional project management activities, including task, budget and cost tracking and forecasting, software project managers are also continuing to be involved in agile projects, taking on facilitating, mentoring, negotiating, protecting and coordinating roles. This is an interesting finding, especially as "*the role of the project manager in an agile project is somewhat of an unknown, because many agile frameworks and approaches do not address the role of the project manager*" (Project Management Institute & Agile Alliance, 2017, p.37).
Finally, the skills involved for being an effective software engineer are different from the skills required to be an effective manager (Malta, 2024). Some managers struggle with management and transition back to being individual contributors (Calbucci, 2020; Hughes, 2022). There appears to be little information about the percentage of managers who transition back or their reasons why.

2.3. Organizational theories

Although many management and organizational theories are relevant, our review of the literature has identified two that we have found particularly relevant to the software design context - an environment of high uncertainty that requires high levels of collaboration. We describe them briefly here and later return to them in the discussion. Lewin (1938) believed that behaviour B was a function of both the person P and their environment E, i.e., $B = f(P, E)$. He referred to this as field theory. Over time, status quos or norms can develop, a condition Lewin called "*quasi-stationary equilibrium*". An individual can have conflicting forces acting upon them. Forces can cause change (or "*unfreezing*") or can work to maintain the status quo. Within a software design context, there are many forces that may change the outcome of the design process. It may be argued that the Agile Manifesto promotes a series of norms, for example, team reflection and self-organizing teams, to improve outcomes in software design projects. The second theory - Ashby's law of requisite variety (1956) - relates to complexity management. Ashby called the number of states in a system its "variety". According to Ashby's law, the control in a system needs to have the same complexity as the system being controlled. When there is complexity (i.e., variety) in a system, such as in the case of software design, one can either try to reduce it (e.g., by limiting changes to requirements) or increase the complexity of the control - "response variety" (e.g., by encouraging self-organizing teams).

3. Method

We report on a series of semi-structured interviews with three managers of software design teams in Waterloo, Canada. These interviews were conducted as part of a larger study surveying software design practice in modern organizations. Participants were known to one of the investigators. Demographic information for the three participants and details about interview dates and length are provided in Table 1. The study received ethics clearance from the institution's research ethics office [#45521].
Two investigators attended all interviews. The primary investigator has more than 20 years of professional experience as a software engineer, software project manager, and software design team manager. This experience informed both the conduct of the interview and the data analyses that ensued. Participants were asked to describe a recent software project that they completed from start to finish, highlighting the design process they followed and design-related documentation artifacts. During the interview, they were prompted to describe what worked and what did not. They were also prompted in some of the areas of the Agile Manifesto, working with customers, the nature of interactions, team measurement, learning, etc. The interviews were conducted and recorded via MS Teams. Transcripts were auto-generated using Otter.ai and then manually reviewed for accuracy. Following Corbin and Strauss (2008), bottom-up coding was completed by the first author using Dedoose, a qualitative analysis software. A total of 59 different codes were identified from 364 excerpts across the three transcripts. These codes were iteratively grouped by similarity into the eight themes that are presented in the next section. For example, "agility" was a theme, with codes related to customer focus, autonomy and doing agile "properly", among others. Another theme - "design" - contained codes related to architecture, expertise, process, representation, etc.

Table 1. Interviewee demographics

	Participant 1 (P1)	**Participant 2 (P2)**	**Participant 3 (P3)**
Role	Project manager and software manager	Manager of managers	Software manager
Team size	7	23 across 3 teams	10-person project team from 5 functional teams
Years in role	4	4	3.5
Years working in software	15	22	19
Industry	Enterprise software	Cloud services	Financial services
Company size	> 100,000 employees	> 150,000 employees	150 employees
Example project	Porting a feature to a new product	Adding a new API	Increasing speed of support tool
Project duration	4 years	1.5 years	8-9 months
Interview length	120 minutes	90 minutes	90 minutes
Interview date	January 2024	February 2024	February 2024

4. Findings

4.1. A not so agile design process . . .

The participants seemed to have a conceptualization of what "proper" agile is but acknowledged that it is different from their way of working. According to P1: "*We're supposed to be an agile company. But I don't think anyone has ever really done agile properly. But . . . in the . . . last six months or so we've sort of done a more iteration-based approach.*" P2 expressed a similar sentiment about agility: "*We don't do like true . . . Agile . . . .whether Kanban or sprints . . . it's kind of like . . . Agilefall, it's like you have timelines, but you still try to like, stick with your work and sprints. We're not great at true Agile.*" P3 expressed strong feelings about agile: "*I hate sprints so much*", preferring to keep administrative overhead low.

It seems the actual design processes followed are highly situated - dependent on the context, organization. As P2 explained, "*so how we do things here is totally different than my last job. And how we do things in my last job was different than the startup before that, where it really was the Wild West.*"

P1 described how even within regular workflows, the process may depend on the nature of the project. In his organization, design relates to changing or adding features to an extremely large and complex software system. For customer-facing software features, a hybrid workflow is followed, where design details are to be approved before implementation is started. However, this standard design process is not used for infrastructure related work, even though the latter is more difficult.

In the case of P2, the high cost of mistakes has imposed a structured design process, with gates acting as "*forcing functions*". There is a large focus at removing ambiguity at the beginning of the process through proof of concepts. P2 described greater success managing using milestones rather than using sprints. He described an organizational "*tree type of hierarchy*" where each sub-group has a technical lead responsible for delivering as aspect of the project:

> *"Within my teams, you know, I think there's more fluidity to how they, you know, manage their work. So, I tend to lean towards giving them more autonomy . . . so [I] just like let them figure out how they want to self-manage, as long as I set the goals and you know, set them up on a path for success. I trust that they will bring things to me when, when I need to be involved."* (P2)

P3 described a project related to the optimization of an internal tool. He approaches design in a highly incremental way and was excited to try his own innovative design process on this project based on iterative loops of "*measure first*" and "*go & see*": "'*Go and see' is just this idea that we can find out the most important issues and what's really going on by just going and watching the work being done.*" "*Measure first*" relates to measuring improvements made

> *"We had learned from trying to understand customer behaviour . . . that the go and see was very effective at finding the actual opportunities, often . . . staring at dashboards or imagining what you think what the problem was, was dramatically less effective than just investigating actual customer experiences." (P3)*

Interestingly, none of the interviewed participants described working directly with external customers. P1 works closely with product management, and P2's team and P3 wors closely with internal customers All three participants described using a JIRA-like issue tracking system for their team's tasks.

Particularly interesting was some of the language participants used to describe their experience. P2 described his organization's design process as "*arduous*" even though the "*business logic is often very simple*". P3 described his as "*boring*" and the management of its participants "*exhausting*".

4.2. Communication

The participants describe different patterns of communication in their organizations. P1 described some uncertainty around how to use meetings most constructively: "*The [meetings] that I found effective are usually like highly collaborative ones, where [] you're trying to come to a decision, you're trying to debate something, you need whiteboard, etc. Less effective ones tend to be status meetings*." According to P2, "*I don't want any meetings. I don't like meetings.*" P3's team held daily standups. He described "*broad team syncs*" that had more of a social function than a coordination function which were used to celebrate team successes. He described "*constraint-based thinking*":

> *"I usually want to be focusing on [what] is the thing that's constraining the team's performance . . . I'm actually very bothered by meetings where you will go over every single task that everyone is working on . . . and talk about how it's going because I think a lot of the things are going just fine. And you know, you really want to spend your time on the thing that is not going fine. And it's going to mess everything up. I forgot I was so passionate about that."* (P3)

Knowledge management was described as a major challenge, particularly when key developers leave. P1 described a video that was made by a developer who left the company, describing the key design decisions that were made on a particular feature. Unfortunately, according to P1: "*what makes sense to you when you're making these videos doesn't make sense three years later.*" The software is now so large and complex that it is not fully documented anywhere. While the format of design specifications is pre-specified, "*nobody likes writing design docs*". P1 noted:

> *"While you're writing [the] design doc, there's always a little bit of coding that needs to happen [to] figure out how it's going to work and what it's going to look like . . . so I do allow my employees to do that. Just with the big caveat that you shouldn't go too far down the development path because inevitably, people are going to disagree with your design and are going to make you change it . . . I find . . . you want to do mock ups of the UI [but] I find it's often easier to just implement it." (P1)*

P2 also described challenges of knowledge management, for example keeping documentation up-to-date and understanding whether a particular problem had been solved before. Unlike P1's organization, P2's was still endeavouring to fully document the design. Designers are given some flexibility in the format of design specifications, meaning that either plain language or diagrams may be used.

P3 described an emphasis on creating shared mental models about how the system worked rather than creating detailed documentation:

> *"There would have been lots of documentation about different aspects of [the system]. I think . . . other engineers had looked at this problem before . . . . And so there were documents floating around that describe, like, you know, their take on different aspects of it. [The design documents] were more about thinking through the approach . . . the design doc acts to sort of like, align the team on the approach. And so everyone understands why they're doing what's being done." (P3)*

Overall, it seems that design documents do not have a binding role in the design process. P3 indicated that formal approvals (that would rely on these documents) were not required to proceed with designs unless "*it's a very serious decision*". P1 also described bottlenecks at design sign-off points, causing "unapproved" design implementations to proceed. Changes to designs could even happen after implementations were demonstrated to stakeholders.

4.3. Design challenges

P1 described numerous design challenges: a large, complex code base that is not well understood, technology changes that require features to be ported, features designed in non-modular ways, etc. P1 endeavoured to reduce technical debt by ensuring that people maintained their own code. Even with his years of experience, he described the challenge of correctly estimating effort.

P2 described the "*long tail*" of unexpected challenges when new features are deployed in a very complex system: "*You kind of play whack a mole at the end when you're trying to like land your product.*" To cope with this uncertainty, he said that "*you push the timelines out really far.*" It was "*hard to iterate fast*" when dealing with software systems of such size and complexity.

The project described by P3 was entirely related to improving the performance of a previous design. P3 used discrete event simulations to model different aspects of the design process. His modeling showed that various factors could cause significant delays in design progress: approvals for pull requests (e.g., design reviews) and meetings. He modified his management approach based on his findings, for example by encouraging his team to "*clear their calendars*" and by making more people available to review code.

For all participants, the design tasks described were related to adding or optimizing functionality in very large software systems. The challenge then was to sufficiently understand the design in order to effectively make those changes. Both P1 and P3 described challenges related to technological changes and the deprecation of older systems.

4.4. Learning and expertise

The three participants also spoke about the importance of expertise in their teams and their approach to increasing it. P1 prefers putting two people on challenging tasks so that they can learn from each other. He also described how his team tried to incorporate best practices, for example when an employee had suggestions about how to be more agile.

P2 said that his teams did formal retrospectives, but that they had struggled to find a useful format for these lessons-learned sessions. P2's hiring process specifically focuses on finding candidates who can handle uncertainty well. "*I'm responsible for their career and performance*". He endeavours to find growth opportunities for his team.

P3 described taking learnings from the marketing team and applying it to his "*go & see*" design methodology. He also complained that language-specific expertise was largely undervalued in the hiring process, noting that developers with extensive experience in a particular language had greater success. As managers, all three participants seemed to have developed expertise related to breaking down work into sub-tasks, though none of them articulated how they did it.

4.5. Organizational challenges

P1 described his organization as follows:

> *"There's [a] big ether out there of things ... you have an idea of what ... happens, but you have no idea [how] to find out ... who to talk to about this, because there's just all kinds of out-of-date documentation, people move teams and that sort of thing"*. (P1)

Prioritization of work is an ongoing challenge because priorities change and it can be challenging to optimize task assignments. P1 described processes prescribed by his company that he avoids because he doesn't think they add any value. P2 described the challenges of meeting organizational timelines:

> *"But what doesn't work for us, I think is like, kind of timeline driven development. I know what our goals are, but things just run into all kinds of different issues... I understand the need to have timelines because if you don't have timelines, then well you know, just like meetings, the projects take the maximum amount of possible time possible until they finish. But at the same time, it just doesn't work"*. (P2)

P3 described numerous organizational challenges including prioritization of resources, and a culture which valued some work more than others. He felt like a "*technician*" of the process, having insufficient influence over some aspects of management. He described himself as a "*systematizer*" who found it tiring to think about people's feelings all the time: "*So I would sort of get by more on like, technical strength than interpersonal sort of intuition. But eventually... it wasn't really how I wanted to spend my days.*" He described problems related to how the company was organized, focusing more on customer-facing features rather than internal efficiency. Like P1, P3 noted: "*I just worked within my sphere of control. ... And then I would interface with the existing processes otherwise, but within the team, I could sort of do what I wanted to do.*"

4.6. Measurement challenges

Both P1 and P3 described the challenges of evaluating software developers. P3 described the topic as "*stressful*"; P1 described the evaluation of developers as "*tough*". According to P1 "*without really like watching them 24/7, it's hard to really know their progress*", particularly when they are new. P1 admitted that "*it comes down to the feeling... it's really hard to get metrics for evaluating people.*" P3 said:

> *"I have thought a lot about that, actually. But ... I haven't really figured that one out.... The problem... with like, measuring people's performance is that you don't know what is coming from their individual contributions versus sort of external noise in the outcomes."* (P3)

P2 and P3 described the use of performance measurement rubrics. According to P3:

> *"A big part of what I would do as an engineering manager is help people sort of grow in their careers, and, you know, make more money, basically, by providing a greater amount of value to the company and our customers ... I would try to set up their work so that they would end up producing these artifacts that I could use as evidence of that they were demonstrating these criteria."* (P3)

4.7. Culture and norms

P1emphasized inclusiveness in trying out different ideas. He also emphasized a culture of code ownership: "*it's our team's responsibility to own the code and keep it good.*"

P2 put a lot of effort into the team cohesion and the development of psychological safety. One of his major objectives is to build community: "[I need to] *make sure there's a good environment for people to succeed in, make sure that people are connected, make sure that you know, not just my team, but community can also mean my wider organization.*"

P3 worked hard to overcome what he viewed as a demotivating culture around working on non-customer facing projects. He sought alignment amongst like-minded people who were excited by the work his team was doing: "*I think that's just the biggest thing ... to match people with the work that they enjoy.*"

7.8. Tools

All three participants expressed enthusiasm about the use of AI to increase productivity. AI was described as being used for code completion, simplifying code, code reviews, creating unit tests, etc. P3 noted that the strongest programmers were most excited to use it. He described expected "*winner take all sort of effects*" with respect to the power of these tools in the hands of expert programmers. With respect to AI usage in the future, P1 said:

> *"Coding is more than just writing the code. [It] is ... solving the problems, knowing ... what you want things to look like in the code, how you want things architected and changed in the code. So that's different than architecting the feature itself."* (P1)

None of the participants mentioned using AI to help measure the people or the product.

5. Discussion

This study was motivated by a need to better understand software design process and its management in modern organizations. The three software managers interviewed in our study were very thoughtful and contemplative in their approach to management, and in the lengthy interviews, openly shared their experiences and perspectives on the management of software design in their organizations. While findings are based on just three interviews, the small sample size does not in itself limit their usefulness to inform future theory development (see discussion in Nickel et al. (2024), pp.19-21).

The first finding relates to the relationship between the participants' software design processes and agile methods. While all three were aware of agile and were following it to some degree, their processes required stages and milestones. We saw distinct design "phases", particularly in the design processes of P1 and P2. In the case of P2, this appeared to be driven by the high cost of mistakes. Other important aspects of agility relate to customer interaction and communication. Generally, we did not see emphasis on usability or customer interaction in these interviews, though we did see close interaction between different internal teams and with "internal" customers. Information exchange and knowledge management are clearly core design activities and are the basis of many challenges. Highsmith and Orr's (2013) model of speculate, collaborate and learn appears to have been reflected somewhat in the design processes described by all participants, particularly that of P3. P3 collaborated closely with his colleagues to create a shared understanding of problems and emphasized measurements as a way of learning. P1 and P2 were more plan-driven than might have been anticipated by the model, but did emphasize collaboration and learning in their teams. For P2, speculation was seen in the proof of concepts at the beginning of projects. We saw a more structured approach than may have been anticipated by Ralph's (2016) model. We saw sensemaking (e.g., making sense of the project context) as P1 develops code to understand the design while writing design documentation, as P2 develops proof of concepts and as P3 refines shared mental models through conversation. We saw different roles and ways of working (e.g., different communication patterns, different ways of documenting design) in each organizations, consistent with Petre et al.’s (2010) assertion that there is no single way to consider the software design process.

An interesting insight emerges as we relate the findings to Brook’s (1987) theory. The interviews seem to show that the effort required for the “accidental” part of design (Brooks, 1987) - coding - may still be quite large, as developers struggle to change increasingly complex code bases. This finding challenges Brooks’ (1987) prediction that we would see productivity gains on this aspect of design. Nevertheless, such gains could be soon realized with the help of AI. Participants believed that AI was increasingly helpful with basic coding tasks, but did not believe that it was yet useful for tasks related to “essential” aspects of software design, such as feature architecting. Similarly, as we relate findings to Gero and Kannengiesser’s (2007) work, key challenges for these practitioners seem to be at the level of “structure”, rather than “function” or “behaviour” due to the complexity of the software.

In all three interviews we saw examples of reflection, whether it was P1’s personal reflections on code ownership, P2’s formal team retrospectives or P3’s attempts to try a new design process. These examples are in line with Schön’s (1983) ‘situated cognition’ approach to design, a learning process whereby designers devise and reflect upon experiments as they go, iteratively improving their work. After the design activity, designers may reflect on what has transpired, an activity Schön referred to as reflection-on-action. Of note across interviews is the way that P1 adapts the system to make it work for his team or the way that P2 and P3 do what they can within their own spheres of control. These approaches could be viewed as a form of “muddling through” (Lindblom, 1959). Lindblom’s theory was based on the core acknowledgement that humans cannot make fully informed decisions due to complex situations and bounded rationality. As such, Lindblom proposed an exploratory, incrementalist approach whereby repeated, small improvements were made with limited analysis. Recently, Norman and Stappers (2015) discuss the merits of revisiting theory of ‘muddling through’ in situations of high complexity. Although they were not specifically discussing software design contexts, comparisons can easily be made between “muddling through” and incremental agile software project management approaches.

In summary, we have a design team, their complex software and their complex context/environment, both internal (the organization) and external (e.g., customers, technological change, etc.). We have a process dominated by information exchange between stakeholders, but this information exchange is imperfect and very context dependent. Different patterns of communication are seen in each of the organizations and a lot of time is spent communicating. Despite this, design knowledge appears difficult to share and keep up-to-date. Complexity and challenges of communication seem to exacerbate one another over time. The software becomes more complex, which means that it is harder to document and reason about. Consequently, sub-optimal design changes may be made, thus increasing the design complexity. We have Lewinian-style (Lewin, 1938) “forces” bombarding the design team from many directions, including technological change, changed priorities, corporate processes and surprise design issues. The software design manager thus operates within an internal and external environment of high degree of uncertainty and complexity, acting as an “excess variety” handler (Ashby, 1956), for example by reprioritizing work, creating a learning environment, promoting particular norms and generally finding ways to move forward by adapting the design process for their own needs. A key mechanism for coping with this context appears to be expertise, both with respect to software design and with respect to the specific context in which it occurs. Expertise requires learning, so it is interesting to see the emphasis on learning in each of the interviews.

6. Contribution, significance, and future research

This paper aimed to answer previous calls (Hall & Rapanotti, 2017; Petre et al., 2010) to better understand modern software design practice and its management within organizations. While software design is recognized as distinct from other forms of engineering design, research on its management and underlying theories remains limited. With respect to software design management, this research fills a gap by looking at management from a design (and not just a project management) point of view. From the perspective of a general theory of software design, it attempts to bring together various theories and offer an empirical view of their applicability in modern professional practice. Overall, our findings highlight the need for future research to better understand what mature software design practice should look like, and for a more design-centred lens to studying software development and management.

References

Adolph, S., & Kruchten, P. (2011). Reconciling Perspectives: How People Manage the Process of Software Development. *2011 Agile Conference*, 48–56. https://doi.org/10.1109/AGILE.2011.43

Ambler, S. (2023). What Happened to the Rational Unified Process (RUP)? https://scottambler.com/what-happened-to-rup

Ashby, W. R. (1956). An introduction to cybernetics. Chapman & Hall.

Beck, K., & Al., E. (2001). Agile Manifesto. https://agilemanifesto.org/

Brooks, F. P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4), 10–19. https://doi.org/10.1109/MC.1987.1663532

Brooks, F. P. (2010). The Design of Design. Addison-Wesley Professional.

Calbucci, M. (2020). 17 reasons why becoming an engineering manager is not what you thought it will be. https://calbucci.com/17-reasons-why-becoming-an-engineering-manager-is-not-what-you-thought-it-will-be-19721f13cb60

Corbin, J., & Strauss, A. (2008). Basics of Qualitative Research 3e. Sage Publications.

Davis, A. M. (1995). 201 Principles of Software Development. McGraw-Hill.

Endres, A., & Rombach, D. (2003). A Handbook of Software and Systems Engineering: Empirical Observations, *Laws and Theories (T. F. I. S. on S. Engineering (ed.))*. Pearson Education Limited.

Evans, M. (2023). Engineering manager's handbook: an insider's guide to managing software development and engineering teams. Packt Publishing Ltd.

Garvin, D. A. (2013). How Google sold its engineers on management. *Harvard Business Review*, December.

Gero, J. S., & Kannengiesser, U. (2007). An Ontological Model of Emergent Design. *International Conference on Engineering Design (ICED).*

Hall, J. G., & Rapanotti, L. (2017). A design theory for software engineering. *Information and Software Technology*, 87, 46–61. https://doi.org/10.1016/j.infsof.2017.01.010

Highsmith, J. A., & Orr, K. (2013). Adaptive software development: a collaborative approach to managing complex systems. *Dorset House Publishing*.

Hughes, K. (2022). What you give up when moving into engineering management. https://stackoverflow.blog/2022/02/23/what-you-give-up-when-moving-into-engineering-management/

Johnson, P., Ekstedt, M., & Jacobson, I. (2012). Where's the theory for software engineering? *IEEE Software*, 29(5), 94–95. https://doi.org/10.1109/MS.2012.127

Kalliamvakou, E., Bird, C., Zimmermann, T., Begel, A., Deline, R., & German, D. M. (2019). What Makes a Great Manager of Software Engineers? *IEEE Transactions on Software Engineering*, 45(1), 87–106. https://doi.org/10.1109/TSE.2017.2768368

Kannengiesser, U., & Gero, J. S. (2015). Is designing independent of domain? Comparing models of engineering, software and service design. *Research in Engineering Design*, 26(3), 253–275. https://doi.org/10.1007/s00163-015-0195-y

Lewin, K. (1938). The Conceptual Representation and the Measurement of Psychological Forces. Duke University Press.

Lindblom, C. E. (1959). The Science of "Muddling Through." Public Administration Review, 19(2), 79–88. https://doi.org/10.2307/973677

Malta, M. (2024). Software Engineering Manager vs. Individual Contributor: The Career Dilemma. https://blog.covibe.us/charting-your-path-the-software-engineering-manager-vs-individual-contributor-dilemma/

Nickel J, Hurst A, Duimering PR. (2024) Contextual influences on trade-offs in engineering design: a qualitative study, *Design Science*, 10(e21). https://doi.org/10.1017/dsj.2024.34

Norman, D. A., & Stappers, P. J. (2015). DesignX: Complex Sociotechnical Systems. *She Ji*, 1(2), 83–106. https://doi.org/10.1016/j.sheji.2016.01.002

Petre, M., van der Hoek, A., & Baker, A. (2010). Editorial. *Design Studies*, 31(6), 533–544. https://doi.org/10.1016/j.destud.2010.09.001

Project Management Institute, & Agile Alliance. (2017). Agile Practice Guide.

Ralph, P. (2016). Software engineering process theory: A multi-method comparison of Sensemaking-Coevolution-Implementation Theory and function-behavior-structure theory. *Information and Software Technology*, 70, 232–250. https://doi.org/10.1016/j.infsof.2015.06.010

Shastri, Y., Hoda, R., & Amor, R. (2017). Understanding the roles of the manager in agile project management. *ACM International Conference Proceeding Series*, 45–55. https://doi.org/10.1145/3021460.3021465

Shastri, Y., Hoda, R., & Amor, R. (2021). The role of the project manager in agile software development projects. *Journal of Systems and Software*, 173(December). https://doi.org/10.1016/j.jss.2020.110871

Wieringa, R. J. (2014). Towards middle-range usable design theories for software engineering. *3rd SEMAT Workshop on General Theories of Software Engineering, GTSE 2014 - Proceedings*, 1–4. https://doi.org/10.1145/2593752.2593753