Aug 10th, 12:00 AM

All Together Now: A Framework for Research on Mob Programming

Wallace Chipidza
Claremont Graduate University, wallace.chipidza@cgu.edu

Watanyoo Suksa-ngiam
Claremont Graduate University, watanyoo.suksa-ngiam@cgu.edu

Mark S. Kim
Claremont Graduate University, mark.kim@cgu.edu

Frank H. Moss
Claremont Graduate University, frank.moss@cgu.edu

Kimani Mbugua
Claremont Graduate University, joseph.mbugua@cgu.edu

See next page for additional authors

Presenter Information

Wallace Chipidza, Watanyoo Suksa-ngiam, Mark S. Kim, Frank H. Moss, Kimani Mbugua, Lorne Olfman, and Terry Ryan

All Together Now: A Framework for Research on Mob Programming

Completed Research

Wallace Chipidza
Claremont Graduate University
wallace.chipidza@cgu.edu

Watanyoo Suksa-Ngiam
Claremont Graduate University
watanyoo.suksa-ngiam@cgu.edu

Mark Kim
Claremont Graduate University
mark.kim@cgu.edu

Kimani Mbugua
Claremont Graduate University
joseph.mbugua@cgu.edu

Frank Moss
Claremont Graduate University
frank.moss@cgu.edu

Lorne Olfman
Claremont Graduate University
lorne.olfman@cgu.edu

Terry Ryan
Claremont Graduate University
terry.ryan@cgu.edu

Abstract

Mob programming (MP) is a relatively new phenomenon in software development. So far, the academic literature has not published in this domain. The goal of this paper is to develop a framework for researching the MP phenomenon. We first outline current practitioner descriptions of and justifications for MP. We then examine concepts from team theory and practices from agile development to identify the team processes and taskwork involved in MP. Based on these, we present a research framework for the academic study of MP. We conclude by considering how the framework can be used by IS researchers.

Keywords

mob programming, mobbing, agile development, software development, teams

Introduction

The history of software development (SD) is littered with innovations. Some of them today seem to have been fads, while others seem to have become accepted practices. As an example of a fad, CASE (Computer Assisted/Aided Software Engineering) tools, once ballyhooed, now generates considerably less 'buzz' than it once did. In contrast, Scrum, once merely a marginal SD method, has become commonplace. Figure 1 shows Google Search results for "CASE tools" compared with "Scrum" from January 2004 to May 2019. Interest in CASE tools declined over this 15-year period while interest in "Scrum" increased steadily.

Recently, a few practitioners in the agile SD community have touted a new SD method, claiming it can lead to dramatic results. The innovation has been dubbed "mob programming" (MP). MP is an agile methodology in which SD teams work "together on … code at the same time, in the same space, and on the same computer" (Shiraishi et al. 2019, p. 616). It is similar to pair programming, except that in MP team size is greater than two, and work is done sequentially by the whole team as they each take turns on the keyboard (Balijepally et al. 2017). To date, evidence for MP claims has been anecdotal, gleaned largely from experience reports at agile practitioner conferences. The reports mention a variety of benefits from the use of MP: higher productivity, higher code quality, lower error rates, increased developer happiness—even reduction in carpal tunnel syndrome.

So far, in Information Systems & Technology (IS&T) research, little has been published about MP.

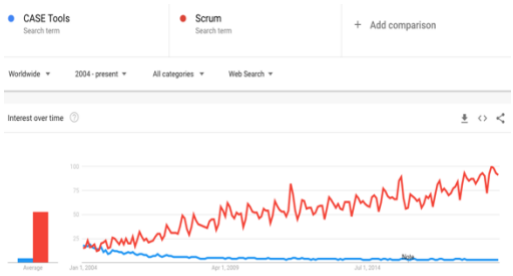

Figure 1: Google Trends interest level:

CASE tools vs. Scrum

This paper proposes a research framework that can be used for the scholarly study of MP. Research frameworks are valuable for discussing and observing phenomena, as well as asking the right questions, organizing findings, and beginning to theorize about them (Gregor 2006). Frameworks can be as minimal as listings of drivers (and variations of these) for phenomena of interest (Nagarajan 2018). They also can embody efforts to relate concepts, research, and theories to advance and systematize knowledge (Rocco and Plakhotnik 2009). They can serve as "organized structures of ideas and concepts," and they can illustrate "relationships between relevant dimensions of a given phenomenon" (Mwansa 2015, p. 109). Thus, researchers may use our framework to formulate useful research questions, derive the right constructs, hypotheses, and levels of analyses for investigating the research questions concerning MP.

A Practitioner View of Mob Programming

According to Woody Zuill, the acknowledged founder of MP, and Kevin Meadows, MP is a software development approach where the whole team works on the same thing, at the same time, in the same space, and at the same computer. This is similar to pair programming, where two people sit at the same computer and collaborate on the same code at the same time. However, with MP, collaboration extends to everyone on the team while still using a single computer for writing the code and doing other work. In addition to coding, the team collaborates to do almost all the work a typical software development team tackles, such as defining user stories, designing, testing, deploying software, and working with the customer, or Product Owner. Almost all the work is handled as "working meetings" or workshops, and all the people involved in creating the software are considered to be team members, including the customer/product owner. (Zuill and Meadows 2016, p. 4).

From this initial definition of MP, one can see that it is a way to do entire SD projects, not merely a way to write code. MP is "an evolutionary step … everyone on the team collaborates with one computer for coding and other work" (Zuill and Meadows 2016, p. 4). It amounts to "all the brilliant minds working on the same thing at the same time in the same place on the same computer" (p. 9). The 'rules' for MP, so far, are quite flexible. Table 1 presents a list of ideas that have worked well for beginning to use MP. As teams adopt MP more completely, many of these ideas are retained.

The "Driver/Navigator model," listed in the Table, involves sharing the roles coder (Driver) and designer (Navigator) across all members of the SD team. The model requires that "[f]or an idea to go from someone's head into the computer it must go through someone else's hands" (p. 16). The driver writes the code, the navigator says what code to write, and the remaining team members watch and listen carefully, contribute ideas, and engage in continuous code review. Zuill and Meadows note that since everyone reviews the code continually, it is better code (p. 61). The idea, "Work on an exercise," in the Table refers to the approach that Zuill and others have used to get organizations started with MP, having programmers try it out on training exercises, rather than on real work. "Do it weekly: for 2-3 hours, or daily for 1-2 hours," describes how long MP sessions should be while trying it out. If the teams like MP, they can adopt it for more frequent and extended periods. Regarding how members of MP teams should work together, the general rule of "Try to show Kindness, Consideration, and Respect" applies. It is a team posture for interactions that aims to avoid resentments, anger, and unhappiness, and to improve flow (p. 27). The idea, "Hold a short retrospective after each session," involves a commitment of team members to asking themselves, "what went well and we want to do more of it? What did not go well and we want to change it?" Answering such questions in retrospective sessions (Derby et al. 2006) allows the team to adapt MP

to how they want to work, tuning it iteratively. The development of MP by Zuill and his colleagues was influenced by some basic principles, common to agile SD.

Size of team: 4 or 5 people
Attendance and participation are voluntary
Arrange a physical setup that is comfortable, simple to put together and use
Follow the Driver/Navigator model Δ
Use a rotation time of 4 or 5 minutes
Work on an exercise Δ
Do it weekly: for 2-3 hours, or daily for 1-2 hours Δ
Invite anyone interested
Try to show Kindness, Consideration, and Respect Δ
Hold a short retrospective after each session Δ

Table 1. Good Ideas for Starting to Do MP (Zuill and Meadows 2016, p. 20)

(Δ Ideas explained in the text)

Practitioner Justification for Using MP

Zuill and his team were not trying to create a new method or extend pair programming, but they noticed that some things were working well and decided to expand on them (p. 5). The changes they made reduced the amount of work they had to do "related to coordinating the efforts of people working separately… things like meetings, emails, follow-ups, status reports, code merges, code reviews …" (p. 65). "Working well together" reduces the need for activities intended to manage the work. Using MP leads to a reduction in "Management Overhead." Faster completion of stories leads to faster feedback. Having the people who can provide answers to questions be part of the team means less waiting. Having four or more sets of eyes always on the code means little/no need for separate code reviews and higher quality. All of these things mean that fewer traditional meetings are needed (p. 66).

Zuill and Meadows (2016) report that using MP caused some common barriers to productivity to fade away. Bare sufficiency (in requirements) is the "absolute minimum required to enable development and testing to proceed with reasonable efficiency" (Waters 2007, p. 4). Only working on what is to be delivered is a primary goal in lean software development; it is associated with eliminating waste (Ambler 2016). Technical debt (TD) has been defined as "the future costs attributable to known violations in production code that should be fixed" (Curtis et al. 2012, p. 34). Lim et al. (2012) found that practitioners see technical debt being accumulated because of deliberate decisions to trade-off this outcome with practical concerns, such as keeping the customer satisfied.

In SD, thrashing refers to the situation where developers spend too much time doing tasks that do not contribute to project goals, e.g., unnecessary meetings, reading/writing of emails, and reading/writing documentation and reports (Meyer et al. 2014; Zuill and Meadows 2016). These thrashing activities are perceived to negatively impact productivity by disrupting the "flow" of work.

Mobbing potentially ameliorates the effects of the barriers on productivity. Because mobbing mandates physical and/or virtual co-location of developers when working on a project, email communication is reduced, and time spent locating where relevant expertise resides within an organization is also reduced. Further, an interruption that indisposes a team member need not interrupt the flow of the whole team; the team theoretically has enough information to advance the project (Lucian 2017; Obermüller and Campbell 2016). In mobbing, developers work on one story at a time, reducing the chance of duplicate code or misinterpretations of the story. Hence, mobbing potentially reduces technical debt.

To the skeptical observer, MP teams may be an inefficient use of resources. While as of yet, there are no rigorous studies of the economic benefits for team sizes of three persons or more, there are numerous studies of productivity gains of Pair Programming (Alistair and Williams 2001; Arisholm et al. 2007; Hannay et al. 2009). Zuill & Meadows claim that the cost of delivering a user story to production by a mob of five programmers is approximately 9% less than the same user story coded by a single programmer (p. 111). They assert that the savings are realized because of the reduction of bugs (defects) in the code. If fewer bugs are delivered to production, fewer hours are required for rework, retesting, and reintegration into the production system. However, these assertions are based predominantly on informally gathered data. Many practitioners, including Zuill & Meadows, are calling for more rigorous research into MP's productivity gains and economic benefits by the academic software engineering community.

Questions to Ask about MP

Based on our experiences talking with practitioners at Agile and MP conferences, as well as our reading of various practitioner papers, we think that they are interested in answers to practical questions about MP: Does it work? Can it be financially justified? For what kinds of SD projects is it well suited? For what kind is it poorly suited? How many people should be in a mob? Are there people who will not do well in MP work? Are certain types of programming languages more suitable for MP than others? What factors will make MP succeed? What factors will make it fail? Answering these questions in ways that are convincing to practitioners—i.e., that make sense in concrete situations—should be a high-priority goal for MP researchers. We also recognize that academics will wish to pursue other—perhaps more abstract—questions, as well. The framework we propose provides a good structure for both kinds of investigation.

Literature review

Three themes arise from our reading of the existing – mostly practitioner – literature on mob programming. The first theme describes how MP is carried out in practice. At a minimum, MP requires three people who alternate the typist/driver and navigator roles (Shiraishi et al. 2019) and sharing a single screen and keyboard (Wilson 2015). The literature also emphasizes variations from the canonical version described by Zuill and Meadows. For example, whereas Zuill and Meadows propose a mob size 4 – 5 people, others propose 6 – 7 (e.g., Kerney 2016; Noaman 2018) or even thousands (Gabriel and Goldman 2000). Moreover, for Zuill and Meadows, mobbing is primarily for coding, but others suggest other activities; for example, refactoring (Helmkamp 2014) and experimentation (Noaman 2018). Arsenovski (2016) describes his team's mob programming method as a "swarm," which features undefined roles and no estimation. Arsenovski does note that at times the group would break into small groups or pairs, including "keep one in the dark" which was designed to prevent groupthink. These variations in practice are potentially challenging for researchers wishing to investigate the efficacy of MP.

A second theme describes the issues that MP practitioners struggle with, which include interpersonal conflict (Kerney 2016), mismatches in developer experience and expertise (Buchan and Pearl 2018; Freudenberg and Wynne 2018; Helmkamp 2014), and scaling the practice to the larger organization (Lucian 2017). Some suggestions for tackling these challenges include emphasizing the notions of kindness, consideration, and respect to resolve interpersonal conflict and mismatches among developer experience and expertise, and enhanced communication with management to enable scaling within the organization (Freudenberg and Wynne 2018; Lucian 2017; Zuill and Meadows 2016).

The third theme describes the various benefits obtained by adopters of MP. Such benefits include improved productivity, higher developer satisfaction, higher quality code as reflected by the reduced number of bugs (Shiraishi et al. 2019). Hunter Industries, perhaps the foremost proponents of MP, claim that they generated zero production bugs in three years (Lucian 2017). Lilienthal (2017) discusses identifying and reducing technical debt as some benefits of MP. The paper suggests, but provides little evidence for, the idea that mobbing could help reduce technical debt by aiding teams in understanding code and architecture problems. Other practitioners see MP as fostering inclusion for introverts and those on the autism spectrum (Freudenberg and Wynne 2018). Another benefit is the learning that accrues especially to less experienced developers as they spend more time interacting with their more experienced counterparts (Boekhout 2016; Helmkamp 2014). These reported benefits might be inspiring the surging interest in MP by practitioners.

MP viewed through team theory

One view of how teams work that can shed light on how MP operates has been provided by Marks, Mathieu, and Zaccaro (Marks et al. 2001). They say, as do many other team researchers, that teams engage in both teamwork and taskwork. In their model, teamwork occurs as team processes. Team processes organize taskwork to achieve team goals, allowing interdependent work to occur, resources to be used, and outcomes to be achieved. Taskwork has to do with what the team is doing. Teamwork has to do with how they are doing it. "Team processes are used to direct, align, and monitor taskwork" (p. 357).

Marks et al. consider time factors in their model, asserting that goal attainment involves the simultaneous pursuit of multiple goals by team members. "Time is linked to goal accomplishment in temporal cycles of goal-directed activity, called episodes" (p. 359). The duration of episodes depends on the nature of tasks being done, the technology being used, and the manner of work chosen by team members. Teams do different things at different times. Sometimes activities emphasize goal accomplishment, sometimes reflection and planning. The first emphasis happens during action phases, the second during transition phases (p. 360). Table 2 lists the teamwork processes in Marks et al.'s model. Planning and evaluation-related processes occur more frequently in transition phases. Coordination and monitoring processes are more frequent in action phases. Interpersonal processes happen in both transition and action (p. 363).

Transition processes
Mission analysis formulation and planning
Goal specification
Strategy formulation
Action processes
Monitoring progress toward goals
Systems monitoring
Team monitoring and backup behavior
Coordination
Interpersonal processes
Conflict management
Motivation and confidence building
Affect management

Table 2. Teamwork Processes from the Recurring Phase Model of Team Processes (Marks et al., 2001)

MP taskwork

Because MP is an agile method, taskwork in MP consists of the work that gets done in agile development, as discussed below (Abrahamsson et al. 2002; Dybå and Dingsøyr 2008; Jeffries 2015). Table 3 lists agile method tasks.

Project Initialization tasks include agreeing on high-level goals, creating the team, and setting project parameters
Requirements-related tasks include getting functional requirements, creating system-level tests, splitting requirements into coding tasks[1]...

[1] The term 'coding task' refers to the creation of code to satisfy a system requirement. It is to be contrasted with the use of 'task' for higher-level taskwork.

Schedule- and budget-setting tasks include estimating time needed for coding tasks, getting priorities from the cust(Pilone and Miles 2008)omer, and calculating team velocity
Increment design tasks include setting goals for the increment, assigning coding tasks to increments, and creating functional tests for the increment's coding tasks
Iteration (sprint) design tasks include updating progress-tracking mechanisms (e.g., burn down chart), and working with the customer to re-prioritize tasks
Iteration (sprint) completion tasks include choosing a coding task, creating unit tests for it, writing code for it, running the unit tests, revising the code until the tests pass, and refactoring the code to avoid technical debt
Increment completion tasks include integrating code with the existing code base, running the functional tests, holding a retrospective for the increment, determining if new requirements exist, demonstrating code to the customer, and reworking estimates

Table 3. Tasks Done in Agile Methods

Factors That Affect SD Outcomes: How IS&T Researchers View It

The IS&T literature contains various models about SD project outcomes and the factors that influence them (Chow and Cao 2008; Lyytinen and Hirschheim 1987; Poulymenakou and Holmes 1996; Scott and Vessey 2002; Wallace and Keil 2004). They are worth noting here, as they are not based explicitly on teamwork or taskwork; instead, they are based on prior SD research published in IS&T-related outlets.

As one example, Wallace and Keil (2004) propose a lengthy list of SD project risks and their effects on outcomes. They map 53 project risk factors to four combinations of the importance of risk (high or moderate) and perceived level of control (high or low). One example—a risk that is high in importance but low in perceived level of control—is conflict between users. Another example—one that is of moderate importance and perceived to be easily controlled—is frequent turnover within the project team (p. 72). The mapping is interesting, albeit maybe a bit unwieldy to serve in a research framework for MP.

Another example, which may fit better with our purposes here, is offered by McLeod and MacDonell (2011). They maintain that SD project outcomes are influenced by four dimensions: institutional context; people and action; development processes; and project content.

Project content comprises "properties of the software systems project itself, including its dimensions, scope and goals, the resources it attracts, and the hardware and software used in development" Institutional context involves "factors related to both the organization in which software systems development is located and the wider socio-economic environment in which the organization operates." The development processes dimension has to do with "the various activities typically associated with software systems development and deployment, ranging from requirements- determination and standard method-use to the management of change resulting from the implementation of a new software system." The people and action dimension includes the “characteristics, actions, interactions and relationships” of “individuals and groups, who are involved or interested in the software systems development project.". Project outcomes are highly situational, and along with the four influencing dimensions are both multidimensional and interactively complex (p. 4).

McLeod and MacDonell (2011) say that determining whether a project has been successful or not can be problematic, as “the different evaluative criteria used by various stakeholders” are often of a “negotiated or contested nature” (p. 4). They explain that project outcomes, beyond simple notions of success or failure, are of different types and span several dimensions. There are outcomes for development processes and/or products; implementation processes and/or products; and system or whole solutions. These outcomes can involve technical, economic, behavioral, psychological, and political aspects. Their natures can be more-or-less subjective, contested/negotiated, or temporal (p. 10).

Mobs are different from well-defined teams because they work on a single task before moving onto the next (Zuill and Meadows 2016). They are also typically self-directed and eschew authority hierarchies (Balijepally et al. 2017). In the absence of a designated leader, the whole team must share accountability

for the success or failure or the project. This implies that top management support is pivotal for MP to succeed. The self-directed, non-hierarchical structure of mobs and requirement for top management support underscore the importance of institutional context and team processes in influencing project success.

Proposed Framework and Its Potential Value

Based on the preceding discussion, we propose the following framework for research concerning MP (Figure 2). Although high-level, the framework should be useful for designing studies of MP with any of several different conventional approaches to IS&T research, discussed briefly below.

The framework points out that MP researchers ought to address associations and/or causal connections among six categories of variables that will be in play whenever MP is used: institutional context (from McLeod & MacDonell, 2011), project content (also from McLeod & MacDonell (2011)), the SD method used (MP, Scrum, XP…); team processes (from Marks, et al., (2002) and Dinh & Salas, (2017), taskwork (from our review of agile tasks, above), and project outcomes (from McLeod & MacDonell, 2011). The figure shows that some of these classes of variables are likely to have direct effects on others, while some are likely primarily to moderate relationships between other categories.

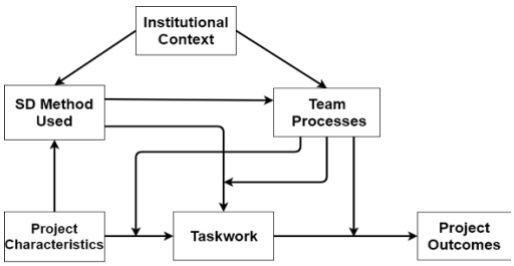

Figure 2: Proposed Research Framework for Investigating Mob Programming

The framework can guide doing various kinds of research about MP. Some interesting empirical questions emerge regarding the value of mob programming. For example, a natural question is: What size would be right for a mob? The literature on team/group size is extensive. Size is one of three key factors, the others being the task at hand and the characteristics of the members of the group. These characteristics include psychological attributes, and leadership ability, and experience with other agile methodologies (Balijepally et al. 2017). Another relevant question is: Can the mob programming approach work in an organization that does not value self-directed teams? Redundancy is a concern of many organizations, but self-directed teams, which are typically redundant, have been found to be effective. The proposed research framework guides answering these questions.

In the case of group size, if it is a characteristic of team processes, then it would likely moderate the relationship between taskwork and project outcomes. A quantitative approach would select a number of projects with similar characteristics but different sizes of mobs, and quantify the nature of taskwork, perhaps in terms of the number of lines of codes, or the number of hours per line of code (or number of lines of code per hour), and the project outcomes such as number of bugs detected or the time taken to resolve bugs, to determine if the project outcomes were moderated by team size (i.e., the magnitude of the interaction effect, or if there is even such an effect). In the case of self-directed teams, a comparative case study could be carried out where two firms with different institutional contexts, one that supports self-directed teams and another that does not. This could be a qualitative study, which uses interviews of organizational members (management and team members) as well as field observations to understand team processes.

Our framework is useful because it is constructed with concepts that have been well studied. Measurement tools exist that can be modified to measure the prescribed framework. For example, “Team Processes” in our framework (Figure 2) draws upon the team theory research by both Dinh and Salas (2017) and Marks et al. (2001). "Coaching," a construct of “Team Processes” has been operationalized in the context of a self-directed team (McLean et al. 2005). Their instrument could be applied in this context. Additionally, new measurements could be developed from the guideline of Moore and Benbasat (1991). New instruments need to be assessed for several types of validity: factorial validity, construct

validity, content validity, convergent validity, discriminant validity, and reliability. This framework can be used to develop causal models to investigate and identify the aspects of MP that may support such claims as that performing taskwork in MP leads to better quality software than in other agile SD methods (Fenton et al. 2002).

Design research could benefit from this framework. Our framework (also known as a model) could be used to inform the design of an IT artifact (Hevner et al. 2004). It could act as the kernel theory for designing and developing software to support mob programming. Models aid problem and solution understanding and frequently represent the connection between problem and solution components enabling exploration of the effects of design decisions and changes in the real world (Hevner et al. 2004, pp. 78–79). The framework could suggest conceptual elements to improve the tools and techniques that mobs use. Action research about MP should be informed by both focal and instrumental theories (Davison et al. 2012). The first kind of theory provides intellectual support for action in MP settings, while the second enables the action research process to be rigorous. The framework we propose above, although more nearly 'theorizing' (Weick 1995) than theory, would help move MP action researchers to these ends. Its inclusion of basic ideas from team theory amounts to an acknowledgment of focal theories from organizational science, and its adoption of taskwork notions from the agile development literature provides the essentials of a theory for analyzing (Gregor 2006) that is more-or-less equivalent to instrumental theory (Davison et al. 2012). Action research projects involving MP could address real practitioner concerns, such as how to introduce MP into different kinds of organizational contexts, while affording IS&T scholars with opportunities to both develop and test theoretical notions about MP.

Our framework potentially benefits experimental research on the efficacy of competing agile methodologies in general and mob programming in particular. The framework is useful for developing hypotheses on the factors influencing various project outcomes. For example, one might hypothesize that the relationship between task complexity and productivity is weaker with MP than with Scrum. The framework also helps identify the theories that should inform the hypotheses (e.g., team theories, institutional theories, and complexity theory), the level of analyses for the experiment, and the populations and context from which to draw the experiment sample. The framework implies that research on agile methodologies should be conducted at the team or organizational levels of analyses. Our framework is beneficial for experimental research because it highlights several factors that must be controlled for in investigating how various factors influence project outcomes, e.g., project characteristics, team size, and institutional context. Using experimentation to test the efficacy of MP would benefit practitioners interested in adopting the methodology. MP suffers from a lack of supporting empirical evidence (Zuill and Meadows 2016), and experimental results may help convince skeptical managers and developers to accept it.

Conclusion

We introduce a new framework for studying Mob Programming (MP). Our literature review only a few academic papers about MP. However, practitioner books, papers, and blogs describe the phenomenon in some detail. We described the current state of the practitioner perspective on MP. Following this, we examined team theory and existing literature on agile methodologies to determine team processes and taskwork for MP. Based on this, we developed a research framework for academic research into MP. We then identified how the framework would be useful to MP researchers using different methods common to the IS field. For example, experiments can be carried out to investigate the effect of project characteristics, mob size, and mob experience on code quality. Design research can be carried out to create software tools to support mobbing activities. We encourage other researchers to attempt to apply the framework.

REFERENCES

Abrahamsson, P., Salo, O., Ronkainen, J., and Warsta, J. 2002. *Agile Software Development Methods: Review and Analysis*, VTT Publications.

Alistair, C., and Williams, L. 2001. "The Costs and Benefits of Pair Programming," *Extreme Programming Examined*, pp. 223–247.

Ambler, S. 2016. "The Principles of Lean Software Development," *Disciplined Agile*, , July 5. (http://disciplinedagiledelivery.com/lean-principles/, accessed June 6, 2019).

Arisholm, E., Gallis, H., Dyba, T., and I.K. Sjoberg, D. 2007. "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise," *IEEE Trans. Softw. Eng.* (33:2), pp. 65–86. (https://doi.org/10.1109/TSE.2007.17).

Arsenovski, D. 2016. "Swarm: Beyond Pair, beyond Scrum," Experience, Agile Alliance Experience Reports, Experience, Agile 2016. (https://www.agilealliance.org/resources/experience-reports/swarm-beyond-pair-beyond-scrum/).

Balijepally, V., Chaudhry, S., and Nerur, S. P. 2017. *Mob Programming - A Promising Innovation in the Agile Toolkit*, presented at the Americas Conference on Information Systems.

Boekhout, K. 2016. "Mob Programming: Find Fun Faster," in *Agile Processes, in Software Engineering, and Extreme Programming*, Lecture Notes in Business Information Processing, H. Sharp and T. Hall (eds.), Springer International Publishing, pp. 185–192.

Buchan, J., and Pearl, M. 2018. "Leveraging the Mob Mentality: An Experience Report on Mob Programming," in *Proceedings of the 22Nd International Conference on Evaluation and Assessment in Software Engineering 2018*, EASE'18, New York, NY, USA: ACM, pp. 199–204. (https://doi.org/10.1145/3210459.3210482).

Chow, T., and Cao, D.-B. 2008. "A Survey Study of Critical Success Factors in Agile Software Projects," *Journal of Systems and Software* (81:6), Agile Product Line Engineering, pp. 961–971. (https://doi.org/10.1016/j.jss.2007.08.020).

Curtis, B., Sappidi, J., and Szynkarski, A. 2012. "Estimating the Principal of an Application's Technical Debt," *IEEE Software* (29:6), pp. 34–42. (https://doi.org/10.1109/MS.2012.156).

Davison, R., Martinsons, M., and Ou, C. 2012. "The Roles of Theory in Canonical Action Research," *MIS Quarterly* (36:3), pp. 763–786.

Derby, E., Larsen, D., and Schwaber, K. 2006. *Agile Retrospectives: Making Good Teams Great*, Pragmatic Bookshelf.

Dinh, J. V., and Salas, E. 2017. "Factors That Influence Teamwork," in *The Wiley Blackwell Handbook of the Psychology of Team Working and Collaborative Processes*, John Wiley, pp. 15–41. (https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118909997#page=31).

Dybå, T., and Dingsøyr, T. 2008. "Empirical Studies of Agile Software Development: A Systematic Review," *Information and Software Technology* (50:9), pp. 833–859.

Fenton, N., Krause, P., and Neil, M. 2002. "Software Measurement: Uncertainty and Causal Modeling," *IEEE Software* (19:4), pp. 116–122. (https://doi.org/10.1109/MS.2002.1020298).

Freudenberg, S., and Wynne, M. 2018. "The Surprisingly Inclusive Benefits of Mob Programming," Experience Report, Agile Alliance Experience Reports, Experience Report, XP2018, May. (https://www.agilealliance.org/resources/experience-reports/the-surprisingly-inclusive-benefits-of-mob-programming/).

Gabriel, R., and Goldman, R. 2000. "Mob Software: The Erotic Life of Code," Essay, Essay.

Gregor, S. 2006. "The Nature of Theory in Information Systems," *MIS Quarterly* (30:3), pp. 611–642.

Hannay, J. E., Dyb\a a, T., Arisholm, E., and Sjøberg, D. I. 2009. "The Effectiveness of Pair Programming: A Meta-Analysis," *Information and Software Technology* (51:7), pp. 1110–1122.

Helmkamp, B. 2014. "Mob Refactoring," *Code Climate*. (https://codeclimate.com/blog/mob-refactoring/, accessed November 15, 2018).

Hevner, A. R., March, S. T., Park, J., and Ram, S. 2004. "Design Science in Information Systems Research," *MIS Quarterly* (28:1), pp. 75–105. (https://doi.org/10.2307/25148625).

Jeffries, R. 2015. *The Nature of Software Development*, The Pragmatic Programmers.

Kerney, J. 2016. "Experience Report, via Initiative of Agile Alliance," *Experience Report, via Initiative of Agile Alliance*.

Lilienthal, C. 2017. "From Pair Programming to Mob Programming to Mob Architecting," in *Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies* (Vol. 269), Lecture Notes in Business Information Processing, Springer International Publishing, pp. 3–12.

Lim, E., Taksande, N., and Seaman, C. 2012. "A Balancing Act: What Software Practitioners Have to Say about Technical Debt," *IEEE Software* (29:6), pp. 22–27. (https://doi.org/10.1109/MS.2012.130).

Lucian, C. 2017. "Growing the Mob," in *Agile Alliance 2017*, Orlando, FL, August 7.

Lyytinen, K., and Hirschheim, R. 1987. *Information Systems Failures—a Survey and Classification of the Empirical Literature*, P. Zorkoczy (ed.), New York, NY, USA: Oxford University Press, Inc., pp. 257–309. (http://dl.acm.org/citation.cfm?id=54890.54898).

Marks, M. A., Mathieu, J. E., and Zaccaro, S. J. 2001. "A Temporally Based Framework and Taxonomy of Team Processes," *Academy of Management Review* (26:3), pp. 356–376.

McLean, G. N., Yang, B., Kuo, M.-H. C., Tolbert, A. S., and Larkin, C. 2005. "Development and Initial Validation of an Instrument Measuring Managerial Coaching Skill," *Human Resource Development Quarterly* (16:2), pp. 157–178. (https://doi.org/10.1002/hrdq.1131).

McLeod, L., and MacDonell, S. G. 2011. *Factors That Affect Software Systems Development Project Outcomes: A Survey of Research*, (43:4).

Meyer, A. N., Fritz, T., Murphy, G. C., and Zimmermann, T. 2014. "Software Developers' Perceptions of Productivity," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, Hong Kong, China: ACM Press, pp. 19–29. (https://doi.org/10.1145/2635868.2635892).

Moore, G. C., and Benbasat, I. 1991. "Development of an Instrument to Measure the Perceptions of Adopting an Information Technology Innovation," *Information Systems Research* (2), pp. 192–222.

Mwansa, G. 2015. "Exploring the Development of a Framework for Agile Methodologies to Promote the Adoption and Use of Cloud Computing Services in South Africa," PhD Thesis, PhD Thesis.

Nagarajan, A. D. 2018. "DevOps Implementation Framework for Agile-Based Large Financial Organizations," Master's Thesis, Master's Thesis.

Noaman, A. 2018. *Experimenting with Mob-Programming*, presented at the Agile 2018, San Diego, CA, August 6.

Obermüller, K., and Campbell, J. 2016. "Mob Programming - the Good, the Bad and the Great," *Under the Hood*, , June 1. (https://underthehood.meltwater.com/blog/2016/06/01/mob-programming/, accessed November 15, 2018).

Pilone, D., and Miles, R. 2008. *Head First Software Development*, O'Reilly.

Poulymenakou, A., and Holmes, A. 1996. "A Contingency Framework for the Investigation of Information Systems Failure," *European Journal of Information Systems* (5:1), pp. 34–46. (https://doi.org/10.1057/ejis.1996.10).

Rocco, T. S., and Plakhotnik, M. S. 2009. "Literature Reviews, Conceptual Frameworks, and Theoretical Frameworks: Terms, Functions, and Distinctions," *Human Resource Development Review* (8:1), pp. 120–130.

Scott, J. E., and Vessey, I. 2002. "Managing Risks in Enterprise Systems Implementations," *Commun. ACM* (45:4), pp. 74–81. (https://doi.org/10.1145/505248.505249).

Shiraishi, M., Washizaki, H., Fukazawa, Y., and Yoder, J. 2019. "Mob Programming: A Systematic Literature Review," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)* (Vol. 2), IEEE, pp. 616–621.

Wallace, L., and Keil, M. 2004. "Software Project Risks and Their Effect on Outcomes," *Commun. ACM* (47:4), pp. 68–73. (https://doi.org/10.1145/975817.975819).

Waters, K. 2007. "Agile Principle 4: Agile Requirements Are Barely Sufficient," *101 Ways*, , March 17. (https://www.101ways.com/2007/03/17/agile-principle-4-agile-requirements-are-barely-sufficient/, accessed June 6, 2019).

Weick, K. E. 1995. "What Theory Is Not, Theorizing Is," *Administrative Science Quarterly* (40:3), pp. 385–390.

Wilson, A. 2015. "Mob Programming - What Works, What Doesn't," in *Agile Processes, in Software Engineering, and Extreme Programming* (Vol. 212), Helsinki, Finland: Springer-Verlag Berlin, May 25, pp. 319–325.

Zuill, W., and Meadows, K. 2016. *Mob Programming: A Whole Team Approach.*, Lean Publishing.