Mob Programming – A Promising Innovation in the Agile Toolkit

Full Paper

VenuGopal Balijepally
Oakland University
balijepa@oakland.edu

Sumera Chaudhry
Oakland University
sichaudh@oakland.edu

Sridhar Nerur
University of Texas at Arlington
snerur@uta.edu

Abstract

Mob programming is a new agile practice that is attracting the attention of software community. This paper presents an overview of mob programming, its benefits and risks identified by its proponents and early adopters and suggests areas for future academic research that could help establish its efficacy and theoretical rationale. The paper also presents the results of text analysis done on the extant literature on the subject.

Keywords

Mob programing, mobbing, eXtreme programming, agile development

Introduction

Agile methodologies, which were inspired by the principles outlined in the agile manifesto (AgileAlliance 2001), have gone mainstream, as evidenced by their current popularity among software organizations (Dingsøyr et al. 2012). Scrum and XP are by far the more popular of the agile approaches today, gauging from their widespread adoption and appeal. With the increasing maturity of agile processes and methods in organizations, software teams have been increasingly customizing and adapting the practices that best serve their project contexts and organizational realities. Mob programming is one such novel adaptation that seeks to scale and extend the collaborative process underpinning the XP practice of pair programming to the whole team.

Mob programming or 'mobbing' is a new agile practice capturing the attention of the software developer community. In this approach, the members of a software development team all work together on the same computer on the same problem by taking turns at the keyboard (Zuill 2014). Mobbing evolved from the personal experiences of Woody Zuill, an agile coach and consultant, when working on software projects at Hunter industries (Zuill 2014). Unlike the eXtreme Programming (XP) practice of pair programming (Beck 2000), where two developers work collaboratively together, mobbing involves the entire software team of about 4 to 5 members working together on the same computer at the same time on the software task at hand. The initial reports from early adopters of this practice are quite encouraging (Boekhout 2016; Wilson 2015), which is contributing to its growing appeal.

While pair programming extended the software development task, traditionally done individually, into a collaborative effort involving a pair, mob programming takes it to a completely new level involving the entire team. That is, the entire team is encouraged to work collaboratively on the same task using one computer and at the same time and place. In other words, the work that was traditionally divided among individual developers and performed in parallel is now done sequentially by the whole group, one task at a time. This certainly raises questions concerning the efficacy of this practice in terms of software quality, throughput, and/or developer productivity achieved. While the initial evidence from reports of early adopters attests to the overall benefits of this approach, it needs further empirical validation from both software practitioners and academics. Similar to pair programming, we believe mob programming has its

merits to be considered as a standard practice for more widespread adoption in software teams. With agile methodologies transforming software development into a socio-technical endeavor, mob programming could be a perfect agile innovation that fits the needs of the new 'socially connected' generation of programmers entering the workplaces. While software developers are experimenting and adapting mob programming in their organizations for realizing potential benefits, it behooves the academic community to step in and explore this phenomenon in terms of providing theoretical underpinnings and establishing its efficacy through rigorous empirical validation. In this research, we provide an overview of mob programming and its process, explore its benefits and risks, and identify potential areas for academic research. We hope this study will motivate IS researchers to embark on this journey to explore this novel phenomenon and its underlying factors to help validate and enhance its efficacy.

The remainder of the paper reads as follows. First, we explain mob programming and its process. Then, we explore the benefits and risks highlighted in the current literature, which mainly comprises reports of early adopters. Next, we present the results of text analysis done on the available corpus of literature on the topic. Further, we identify potential areas concerning mobbing and its efficacy that could benefit from academic research. Finally, we present our conclusions.

Mob Programming

To our knowledge, the term 'mob programming' first appeared in the early XP literature (Hohman and Slocum 2002). While sharing the experiences of their team transitioning into XP practices, Hohman and Slocum (2002)) discuss an approach to code refactoring done in groups involving more than two developers. They called it 'mob programming', deriving from the term pair programming that was becoming popular on the heels of agile manifesto (AgileAlliance 2001) and Kent Beck's espousal of XP and its agile practices (Beck 2000). Inspired by the benefits to pair programming (e.g., cleaner code in less time, shared ownership, seamless communication and instant feedback, etc.) reported in early studies (Cockburn and Williams 2001), Hohman and Slocum (2002)) describe how they experimented to develop a process and structure to their weekly hour long luncheon sessions of 'mob programming', where they tried code refactoring. The primary objectives for these sessions included reinforcing the benefits of XP principles and practices such as clean code, shared code ownership, improved test coverage, knowledge sharing concerning coding standards, programming patterns, design decisions, etc. In terms of the actual process, Hohman and Slocum (2002) discuss three roles that they conceived for the persons involved in these sessions--drivers, narrators and mob. A driver is the person at the keyboard coding, compiling and testing the code, while a narrator is the person who wrote the original code and who initially explains the purpose of the code and how it fits with the rest of the application. The remaining members of the team constitute the mob that discuss the code and provide feedback. Hohman and Slocum (2002) highlight how they continuously tweaked the process and the format of these sessions to help realize their stated objectives. Thus, this early form of 'mob programming' was a one-hour weekly routine to reinforce the XP values of communication, simplicity, feedback and learning (Beck 2000), that involved refactoring a sample code in a group setting. However, it was not until Woody Zuill (Zuill 2014) that 'mob programming' has evolved into its current, well fleshed out process for coding and delivering software by an entire group working together all the time.

In mob programming conceived by Woody Zuill (Zuill 2014), the whole team works on the same task together at the same place using one computer, for extended periods, if not all the time. It is left to the team to decide if they would like to engage in 'mobbing' all the time or some time daily or for a few hours once a week. The team not only works together to code, but also to define stories, design, test and deploy software, and to work with the customer or the product owner. The mobbing team works in a workshop environment with team members seated around a PC with a large monitor or a projected screen, which all members could see. Although Zuill suggests certain layouts that could work well for many workplace settings, it is up to the teams to improvise and settle on the layout that works best for them.

Member Roles

Similar to pair programming, mobbing involves the roles of drivers, navigators and partners. Driver is the person performing the software task at the keyboard, whether it is designing, architecting, coding, debugging, testing or creating a user story. Other members act as Navigators to do other related tasks

such as inspecting the code, searching the internet for any information, seeking clarifications on requirements from customers, etc. Customers or Product owners are referred to as 'Partners' and are encouraged to be part of the team on a full time basis or at least be available on a regular basis for shorter durations. A Partner, when present during a mobbing session, would not be just another navigating member, but also may volunteer to take on the role of a driver for demonstrating specific ideas at the keyboard. In fact, the ideas and thoughts of the navigating members are converted into code or other artifacts through the hands of the driving member. Thus, drivers function as the "hands of the team" on the keyboard (Zuill 2014). A Navigator's job is to discuss different ideas, design a solution and guide the driver to implement his/her code. In addition, a navigator constantly inspects and reviews the code to look for errors and to suggest areas where for code improvement. Ideally, Navigators should to do all the thinking and free up the driver to concentrate on coding the solution. There should be free communication among members so that navigators could clearly communicate their ideas to the driver for quick coding/implementation. Each driver may need a different level of navigational support with some requiring detailed instructions, while others needing only short and quick guidance (Zuill and Meadows 2016).

Role Rotation

Team members need to rotate and take turns at the keyboard at regular intervals, say between 5 and 15 minutes, as decided by the team. Shorter intervals are preferable so that the navigators remain focused and alert to quickly step into the driver role. To enforce timely rotation, teams could use timers that beep at set intervals, as long as it is not distracting or irritating to other teams in the vicinity. To discourage members from attempting that one last thing they would like before turning over the keyboard, anytime a driver hits upon a new idea while working at the keyboard, it is best for the member to give up driving and help navigate another member to test out the idea at the keyboard [].

Core Team Values

Extreme Programming underscores communication, simplicity, feedback, courage and respect as values that help foster agility in software teams (Beck 2000). Drawing from those values that relate to interpersonal dynamics (i.e., communication, feedback, and respect), mob programming emphasizes the importance of developers treating each other with kindness, consideration and respect (Zuill 2014). As conversations during mob programming typically involve five, six or more people, the number of interactions quickly escalate. If teams are engaged in mobbing all day, it is not difficult to see how things could quickly turn unpleasant if members do not follow these principles in each of their interactions. Consciously reminding oneself to be kind and considerate to others ensures that other members will voice their opinions and thoughts with a sense of security and freedom. In any discussion, if people are considerate and respectful to each other then it is see each other's point of view knowing that each member is genuinely trying to help the team. It also becomes less difficult to let others win. Mob programming makes it easy to try out competing ideas swiftly to see which one makes more sense so that no member feels excluded (Zuill and Meadows 2016). This also reduces the costs for taking a wrong turn or for pursuing a dead end and thus increases the teams' appetite for experimentation and discovery.

Benefits and Risks of Mob Programming

Potential Benefits of Mob Programming

One the face of it, mobbing appears highly resource intensive and wasteful with, say, five members of a group working on the same software task instead of the five members working independently on five different tasks. What are some possible benefits that these early adopters see which inspires them to keep doing it every day and at all times? Zuill and Meadows (2016) argue that a better way to appreciate the productivity benefits of mob programming is to understand how mobbing teams mitigate the productivity destroyers ubiquitous in non-mobbing teams.

Less managerial overhead – In teams where developers work independently there is always some extra burden that they carry to coordinate the activities of multiple developers. Such managerial overhead includes activities such as emails, meetings, follow-ups, presentations, code mergers, work schedules,

code reviews, performance reviews, etc. When the team members are talking and coding together, there is less need for emails, meetings, performance monitoring, etc. As every piece of code created by the team goes through the live scrutiny of the entire 'mob', it greatly reduced the need for any code mergers or code reviews. Zuill (2014) admits that he and his team at Hunter Industries did not set out to identify and solve the various issues that hamper productivity, but instead were quite observant and tried to accentuate whatever was working. In the process, they saw great reduction in the managerial overhead.

Less Communication bottlenecks – A developer's actual coding work is often hampered when some information or clarification is needed on a user story or a requirement. The 'question queue time'—the wait time to get an answer to a question blocking a developer—could vary widely for different questions. When the queue times are aggregated for all questions that arise in a typical day for a developer and across all team members, the resulting loss of productivity at the project level could be staggering. When blocked from moving further, to stay busy developers typically take up a new task from the pending inventory of work. Such inventory of unfinished jobs are productivity busters, as stakeholders receive no value until each of them is completed and released for use. As and when developers receive answer to their original question, they incur cognitive switching costs to go back and get up to speed on that task. In mobbing teams, the question queue times are drastically reduced if another member of the team has the answer to the question. If the team's Partners (i.e., product owners) show up to mobbing sessions even for few hours every day and are readily accessible to the team for the rest of the day through phone calls, screen sharing or instant messaging, it would drastically reduce question queue times. The work gets completed and delivered faster instead of ending up in the inventory of unfinished work. (Zuill and Meadows 2016).

Less Decision-making – Software teams are required to make several decisions during the course of project. Making people accountable to their decisions is a common strategy used in organizations, which is well intentioned and on the face of it quite appealing too. However, developers are often called to make decisions with incomplete or imperfect information. Amidst a culture of accountability people become reluctant to make such decisions, which leads to dysfunctions and project delays. When decisions made in good faith end up looking not so good later, people feel compelled to defend and pursue them with vigor until things become defenseless. Thus, a well-intentioned commitment to accountability in decision making often leads to resource waste and costly project delays benefiting no one. Mob programming seeks to avoid such dysfunctions surrounding decision making by practicing 'just-in-time' decision-making—i.e., making decisions concerning only the current task. The underlying assumption is that mobbing teams do not need to make decisions ahead of time, but only when actually working on the task involved in the decision. As work is done incrementally, any bad decisions could easily be undone, thereby containing potential damage. With each member contributing to the work as well as to all the decisions involved therein, the fear of accountability that hampers decision making quickly gives way to courage and a culture of experimentation (Zuill and Meadows 2016).

Less waste and doing only what is barely sufficient – Simplicity in design and emphasis on minimizing feature creep and even test bloat is a core agile principle underpinning all agile methods. When mob programming, many such issues quickly fade away without any conscious effort. For instance, the 'one-piece flow' pattern of workflow (i.e., the entire team working on a feature till its completion and delivery to the customer before moving to the next one) ensures that the utility of a just completed feature could be instantly judged from the live feedback received from the mob. This helps steer the team into the next useful feature to work on. The process continues until the team senses the diminishing value of adding additional features and moves on to work on a different product or module. Thus, the team smoothly grows into the 'barely sufficient' zone, thus avoiding wasteful effort (Zuill and Meadows 2016).

Less technical debt – Technical debt refers to the short cuts and quick fixes that are sometimes done during coding for expediency rather than taking the time to code the best solution possible. When technical debt is allowed to grow, it becomes increasingly difficult to make changes or debug the code. Refactoring the code at regular intervals helps repay it. In mob programming, the continuous code inspection and review done by the navigators encourages the team to stick to coding standards and create cleaner code in the first place, thus minimizing technical debt (Zuill and Meadows 2016).

Less thrashing issues in teams – Thrashing refers to the distractions and interruptions that developers face from their work when disturbed by others for help, or called upon to attend meetings, etc. Thrashing makes them less effective in accomplishing their current work. In a mob programming setting, thrashing

fades away as the team gets to decide how to deal with the interruptions. Depending upon the nature of issue involved, either one member takes care of it while the rest continue with the work at hand, or if urgent, the team drops everything and swiftly addresses the issue. Due to the 'group memory' of mobbing teams, disturbances to the team's flow of work from thrashing episodes is much less compared to independently working developers (Zuill and Meadows 2016).

Less politics – In teams where developers code independently, the incentive structures typically favor individual productivity, leaving people in a bind concerning the extent of help they could provide others who may be seeking it, particularly when facing pressures to get their own work done. In mob programming, with the team tasked to collectively deliver the work, the individual developers freed from the pressures of individual accountability, would be more than willing to help others seeking their help (Zuill and Meadows 2016). Answering a question would not be a distraction but an act of kindness that builds social capital and generates future benefits to the individual.

Less need for meetings – As mob programming is done in a conference format, it is like having an all-day meeting where work is getting done while all related issues are continuously being discussed and resolved at the same time. It is not difficult to see how this diminishes the need for other formal meetings and follow ups that would have been necessary to resolve various issues (Zuill and Meadows 2016).

Continuous learning – Given the right environment, mob programming could amplify learning for all the team members concerned, irrespective of their skill levels or coding experience. When developers are working on a task together, with each bringing their skills, ideas and approaches to play in trying to find a solution, it creates boundless possibilities for learning from each other. Such learning could include coding solutions, design patterns, tool usage, keyboard shortcuts, testing and debugging skills, among others. (Zuill and Meadows 2016).

Higher Developer Satisfaction – After some initial adjustment period, developers report very high satisfaction with the overall experience of doing mob programming. Each task completed by the mob team helps reinforce the effectiveness of the approach to the team members (Zuill and Meadows 2016).

Higher Software Quality – Once a team stabilizes into the mobbing routine, they can see substantial improvements in the quality of code produced. As multiple eyes are inspecting and reviewing the code under development, mobbing teams can easily spot and fix several bugs and errors in the code before releasing for testing or deployment. The resultant code should not only have less bugs, but should also be cleaner and hence more maintainable due to better adherence to coding standards possible during mobbing (Zuill and Meadows 2016).

Possible Risks of Mob Programming

While the experience of early adopters' highlights several benefits to mob programming outlined above (Arsenovski 2016; Boekhout 2016; Kerney 2015; Kolchier 2009; Wilson 2015; Zuill 2014), some potential risks could negate these benefits and thus need to be managed for achieving sustainable results (Zuill and Meadows 2016).

Organizational culture – If an organization has rigid top-down decision-making culture, mob programming may not be a useful approach, as mobbing teams need to be self-organizing. They need the latitude to make most decisions within the team to be effective at delivering working code on a daily basis. If a mobbing team of four or five developers face regular disruptions waiting for decisions or answers from stakeholders external to the team, it will be quite wasteful seriously affecting the productivity of the team.

Familiarity with Agile Development – If an organization is not already using agile methods, it will be difficult to realize expected productivity benefits from mob programming. On the other hand, if a software team is already agile and has experience with pair programming, then getting into mob programming would be a minor cognitive leap for the concerned people that drastically reduces overall risk for the team.

Dominant Personalities – If there are some dominant personalities within mob teams and if they are the ones doing most of the talking and are steering the whole team into their ideas and solutions, then others may find the whole team atmosphere less conducive to their own personal development. They may slowly switch off and even dump mobbing altogether to go back to their independent work routines. As signing

up for mobbing should generally be a voluntary undertaking, it is incumbent on all the members to create a hospitable environment for everybody to be able to contribute and feel valued. Treating each other with kindness, consideration and respect in all their daily interactions becomes central to keeping them all together.

Developer Safety and Fatigue – If mob teams have to work together all day, having the right workplace setup becomes essential. Having the right equipment (e.g., comfortable chairs, large monitor or projector/screen, multiple keyboards, whiteboards, speakerphones, hand sanitizers, etc.), and right office space where teams can talk aloud without disturbing other teams, private work areas for developers when not engaged in mobbing, etc. all become critical enablers for creating a safe and productive workspace for mobbing. As mob programming could be quite intensive, developers could experience exhaustion and fatigue when mobbing for extended periods, or for the whole day. Keeping a manageable work schedule with periodic breaks and letting members tend to their personal or other official chores (e.g., filling expense reports or time sheets, attending other meetings, catching up on emails/IMs, etc.) while the rest are mobbing, etc. helps create a sustainable work environment for the developers.

Text Analysis of Mob Programming Articles

In order to get a sense of the dominant themes and keywords underlying the phenomenon of mob programming, we performed text mining on the sparse literature on mobbing. This included seven articles and a seminal book on Mob Programming by Zuill and Meadows (2016). The following steps were followed in performing text analysis:

a) Articles and the book were available in PDF format, so they were converted to text.

b) The text obtained from step (a) was then preprocessed. This included:

   i. converting text to lowercase;

   ii. removing common English stopwords (e.g., is, of, the, a) as well as other words (e.g., finally, conclusion, based) that come in the way of interpreting the results; and

   iii. removing digits and punctuation.

c) The parsed files were then used to generate a word cloud (see Figure 1), to obtain relationships of words based on their co-occurrence frequency (Figure 2), and to extract key themes latent in the text (see Table 1).

The word cloud shown in figure 1 captures many of the key words that have been used to describe mob programming. Words such as “learning”, “driver”, “whole”, “retrospective”, “defect”, “idea”, “keyboard”, “rotation” and “whole” capture the essence of the phenomenon under discussion.

VosViewer, an excellent software package from Leiden University in the Netherlands, was used to visualize words and their proximities based on co-occurrence frequency (Van Eck and Waltman 2011). Figure 2 highlights the prominent words in the mob programming literature.

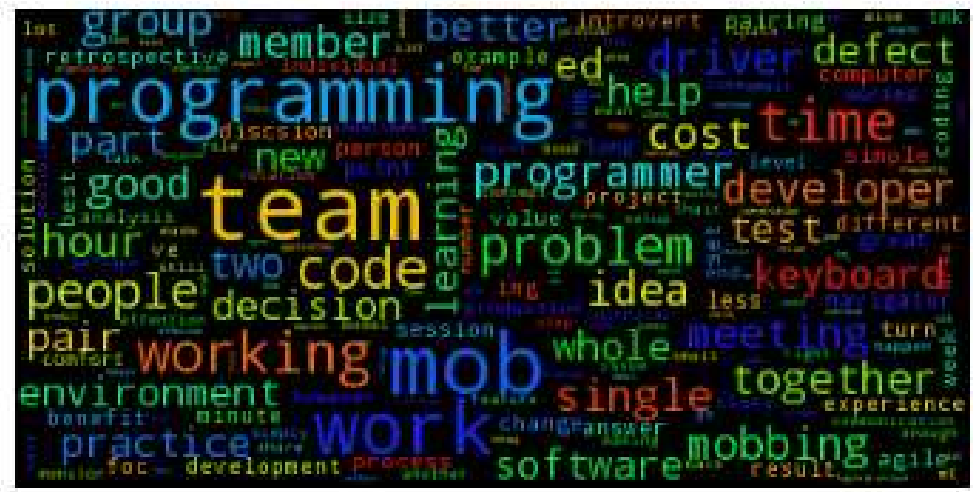

Figure 1. Word Map showing dominant words

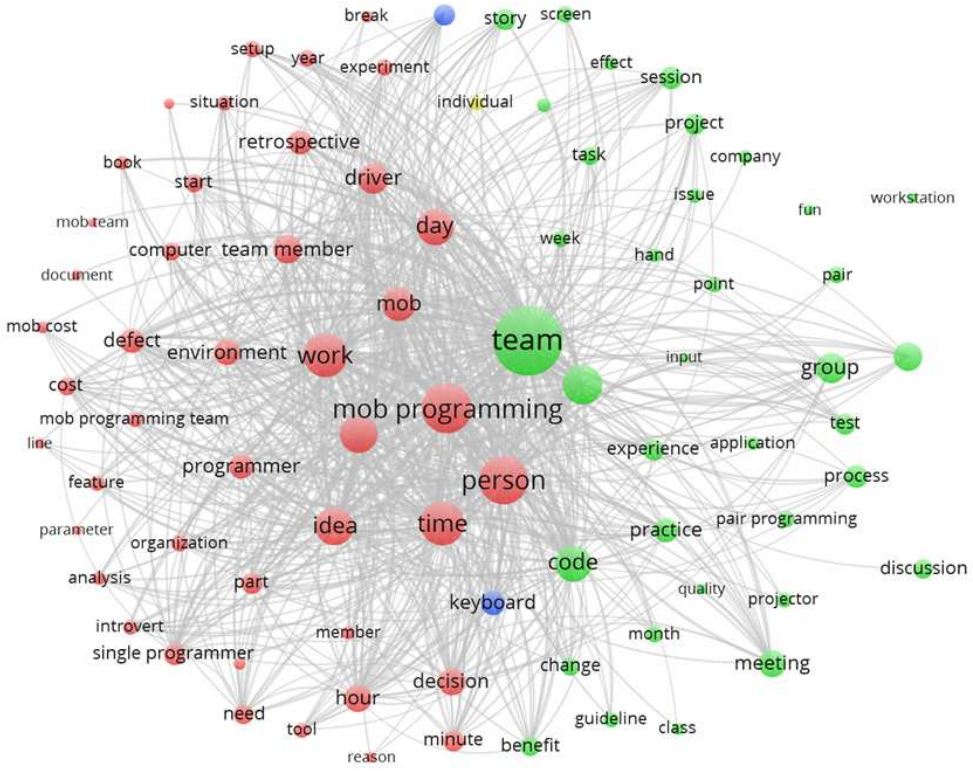

Figure 2. Word Relationships

Finally, we used MALLET (Machine Learning for Language Toolkit) from the University of Massachusetts at Amherst, to extract eight topics (as many as the number of documents being examined). Specifically, the tool uses a popular algorithm called Latent Dirichlet Allocation (LDA) to identify topics and their associated words (McCallum 2002). Table 1 shows the key topics and the most prominent words that define each topic. Evident in these topics are issues related to process (e.g., role, meeting, keyboard,

group), practices (e.g., Test Driven Development), people (e.g., trust, kindness), and benefits (e.g., learning, enjoyable).

Topic No.	Keywords
0	code meetings meeting programming discussion process developers group role people groups format developer practices refactoring current class values extreme design
1	story mobbing teams day process board retro junior experiment room unruly goal red skills session learning developers hourly faster coach
2	computer typical meeting concept keyboards action lean monitors style meaningful comfortable holding projects kindness phone blog manifesto list principles navigators
3	swarming pair swarm thetitleofyourexperiencereport application teams system client roles social decision initiative spreadout server members legacy intense turns enjoy enjoyable
4	cost programmer single problem mobbing defects simple analysis costs hours introvert defect introverts answer size story model debt metrics remote
5	hunter trust benefit timer typing guideline rules kindness job memory financial guidelines week study awesome speaking amazing outcome read view
6	team mob programming work time working code people driver day good learning software problems idea keyboard members ideas important person
7	group pairing developers developer test team development pair driving rubber style strong week code soa tdd ping services pong project

Table 1. Topics and Keywords resulting from the Textual Analysis of Mob Programming Articles

Potential Areas for Future IS Research

As indicated earlier, mob programming is a relatively new agile practice that is catching the attention of the software developers. The book authored by Zuill and Meadows (2016), is the main source for understanding mob programming and how to get started on it. In addition, some early adopters have shared experiences reports that appear in some agile conference proceedings [e.g., (Arsenovski 2016; Wilson 2015)]. The empirical evidence concerning its effectiveness is based on the reports of Woody Zuill and his team members who have more than four years of experience working on it. It is quite noteworthy that all the early adapters are seeing great value and attesting to its benefits. However, it is important that academic community also steps-in to validate its efficacy and explore its theoretical rationale for the benefit of software practice. Towards this end, we identify some research areas relating to mob programming that could benefit from academic inquiry.

Theoretical Rationale – Innovations in software development methods and practices routinely happen in the trenches of software practice, which are then put to rigorous theoretical and empirical scrutiny by the academic community to assure the practitioner world of its efficacy and to caution against potential risks. As Jacobson and Spence (2009) suggest, sound theoretical underpinnings help identify the core 'truths' of software development that are independent of any methodology or framework. As mobbing involves group work, theoretical perspectives from social psychology that explore group processes and relative productivity of groups versus individuals [e.g., (Hill 1982)] when working on various tasks would be a good starting point. It is also important not to lose sight of group dysfunctions such as groupthink, social loafing [e.g., (Harkins 1987)] that could hamper mob team productivity. The holographic principles of organizational design (Morgan and Ramirez 1984) is another theoretical perspective that could help understand the effectiveness of mobbing practices and procedures.

Empirical Validation – While the initial evidence for the effectiveness of mob programming by its proponents and early adapters is quite encouraging, it still requires validation through rigorously designed controlled experiments, so that more organizations could adopt it fully understanding the conditions that accentuate its benefits and minimize potential risks. Conducting such controlled experiments and establishing the statistical significance of resultant findings falls squarely within the expertise of the academic community. If academics could collaborate with software practitioners in this endeavor, it could help establish mob programming as a viable agile practice and speed up its adoption and diffusion beyond its early prospectors.

Other Research Topics – There are several factors concerning mob programming that could impact its efficacy. Such factors, which need further investigation, include the following: team size; team composition in terms of abilities, personality factors and gender; project characteristics such as size, complexity and novelty; organization culture; and collocated versus distributed teams. For instance, we believe mob programming could fit in with the working styles of new socially connected generation of IS/Computer Science graduates entering IT workforce. Thus, it will be interesting to see how age, experience, and social connectedness of developers contributes to the effectiveness of mob programming. The factors and issues identified above are just illustrative and are by means comprehensive.

Conclusion

Software development approaches have undergone major changes in recent times. While there is little doubt that Agile Methodologies are dominant in industry, practices employed by organizations can vary considerably. Most of these practices have been evolved to ensure that: a) there are no communication breakdowns; b) the entire team is on the same page and all the information needed to solve the problem is readily available; c) a truly collaborative environment that fosters a climate of creativity exists; and d) the entire team collectively senses and responds to challenges that emerge. Mob programming is an outcome of the efforts of organizations to achieve the aforementioned objectives.

In this study, we provide an overview of mob programming, outlining the motivation for the practice and the benefits it confers. Furthermore, our paper, drawing on past empirical evidence, articulates the challenges that have to be overcome to make mob programming efficacious. Our review opens up avenues for empirical research on the practice of mob programming.

REFERENCES

AgileAlliance. 2001. "Manifesto for Agile Software Development." Retrieved Jun 1, 2013, from http://www.agilemanifesto.org

Arsenovski, D. 2016. "Swarm - Beyond Pair, Beyond Scrum," *Agile 2016 Conference*, Atlanta, GA.

Beck, K. 2000. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison Wesley.

Boekhout, K. 2016. "Mob Programming: Find Fun Faster," *17th International Conference on Agile Software Development, XP 2016,* H. Sharp and T. Hall (eds.), Edinburgh, UK: Springer, pp. 185-192.

Cockburn, A., and Williams, L. 2001. "The Costs and Benefits of Pair Programming," in *Extreme Programming Examined,* G. Succi and M. Marchesi (eds.). Boston, MA: Addison Wesley, pp. 223-243.

Dingsøyr, T., Nerur, S., Balijepally, V., and Moe, N.B. 2012. "A Decade of Agile Methodologies: Towards Explaining Agile Software Development," *Journal of Systems & Software* (85:6), pp. 1213-1221.

Harkins, S.G. 1987. "Social Loafing and Social Facilitation," *Journal of Experimental Social Psychology* (23:11), pp. 1-18.

Hill, G.W. 1982. "Group Versus Individual Performance: Are N + 1 Heads Better Than One?," *Psychological Bulletin* (91:3), pp. 517-539.

Hohman, M.M., and Slocum, A.C. 2002. "Mob Programming and the Transition to Xp," in *Extreme Programming Perspectives,* G. Succi (ed.). Boston, MA: Addison-Wesley, pp. 323-334.

Jacobson, I., and Spence, I. 2009. "Why We Need a Theory for Software Engineering," in: *Dr. Dobb's Journal*.

Kerney, R.J. 2015. "Mob Programming - My First Team."

Kolchier, K. 2009. "Exploring Synergistic Impact through Adventures in Group Pairing," *Agile Conference, 2009. AGILE'09.*: IEEE, pp. 265-270.

McCallum, A.K. 2002. "Mallet: A Machine Learning for Language Toolkit." from http://mallet.cs.umass.edu

Morgan, G., and Ramirez, R. 1984. "Action Learning: A Holographic Metaphor for Guiding Social Change," *Human Relations* (37:1), January 1, 1984, pp. 1-27.

Van Eck, N.J., and Waltman, L. 2011. "Text Mining and Visualization Using Vosviewer," *ISSI Newsletter* (7:3), pp. 50-54.

Wilson, A. 2015. "Mob Programming-What Works, What Doesn't," *International Conference on Agile Software Development*: Springer, pp. 319-325.

Zuill, W. 2014. "Mob Programming–a Whole Team Approach," *Agile 2014 Conference, Orlando, Florida*.

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