

Delivering Value with Confidence

“Swarming Patterns”

Joseph W. Yoder, The Refactory, Inc. – USA

Danijel Arsenovski, SolutinosIQ – USA

Ademar Aguiar, Faculdade de Engenharia, Universidade do Porto – Portugal

Hironori Washizaki, Waseda University – Japan

Many software development processes such as Agile and Lean focus on the delivery of working software that meets the needs of the end users. Many of these development processes help teams respond to unpredictability through incremental, iterative work cadences and through empirical feedback. There is a commitment to quickly deliver reliable working software that has the highest value to those using or benefiting from the software. A key principle to the long term success of a project is during the development and release cycles, to have confidence that changes will not break important parts of the system. This is only done if there is time taken to build confidence into the process and architecture. Swarming is a technique where multiple people work together to complete one or more tasks. Recently there has been success with techniques such as swarming to help assure the delivery meets the requirements and proper validation and checks are done before release. This paper will focus on some “Swarming” patterns and how they assist teams to Deliver with Confidence.

Categories and Subject Descriptors

- Software and its engineering ~ Agile software development • Social and professional topics
- Software and its engineering ~

General Terms

Agile, Sustainable Delivery, Patterns, Swarming, Confidence,

Additional Keywords and Phrases

Agile Software Development, Mob Programming, Innovation, Reliability,

ACM Reference Format:

Yoder, J.W., Washizaki, H., Aguiar A., 2018. Delivering Fast with Confidence “DevOps”. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 7 (October), 7 pages.

NOTE TO SL PC AND SHEPHERD. WE ARE EVOLVING THESE AND NEED TO GET THE DETAILS OF THE PATTERNS FILLED IN. We have outlined the ideas for now.

Author's email address: joe@refactory.com, washizaki@waseda.jp, ademar.aguiar@fe.up.pt

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 12th Latin American Conference on Pattern Languages of Programs (SugarLoafPLoP). SugarLoafPLoP'18, November 20-23, Valparaíso, Chile. Copyright 2018 is held by the author(s). HILLSIDE 978-1-XXXXXX-YY-Z.

Introduction

Being Agile, with its attention to extensive testing, frequent integration, and focusing on important product features, has proven invaluable to many software teams. However, blindly following Agile practices is not sufficient to help sustain delivering quality software at a good pace with confidence. When building complex systems, it can be all too easy to primarily focus on features and overlook software qualities, specifically those related to the architecture and sustainability. Some believe that by simply following Agile practices—starting as fast as possible, keeping code clean, and having lots of tests—a good architecture will magically emerge. While an architecture will emerge, if there is not enough attention paid to it and the code, technical debt and design problems will creep in until it becomes muddy, making it hard to deliver new features quickly and reliably. It is essential to have a sustainable architecture that can evolve through the project life-cycle. Sustainable architecture requires ongoing attention, especially when there are evolving priorities, lots of technical risk, and many dependencies.

There are many proven practices that are used during development to help sustain regular systematic deliveries while maintaining confidence in the system. Some of these are related to Agile or Lean practices such as short delivery cycles, testing, clean code, and continuous integration. Small delivery size with regular feedback through incremental releases has proven itself over the years and has become the de-facto standard for most Agile practices.

Recently there has been observed success in various swarming techniques to assist with sustaining development with confidence. Swarming can be where only two team members work together on the same task or user story (similar to pair programming). The other end of the spectrum has all team members working on the same user story which is called “mob programming¹”. In the later, all members focus on solving a particular problem by working on tasks together using a single computer.

Every programmer has experienced moments of mental exhaustion where it can take a lot of time to resolve even the most trivial problem. Most programmers (even those that do not practice pair or mob programming) are aware of this and will find ways to take breaks and invite fellow programmers to take a look at their code in those moments of frustration. In mob programming sessions, these moments are practically non-existent. Mobbing is a form of swarming where the team works collectively to solve problems, releasing software regularly.

The patterns presented in this paper are used during swarming to address common issues: *Branch Out*, *Spread Out*, and *Keep One in the Dark*. These patterns support swarming and provide the benefit for the team to react and reorganize allowing the main group to continue working on current items in “business as usual” fashion, while a few members would deal with impediments and urgent or unplanned issues. This means that the normal team dynamic remains unaffected, learning continues and all members are still very much aware of the current work in progress. This paper will focus on the “Swarming” patterns as a practice to help sustain “Delivering Value with Confidence”.

¹ "Mob Programming – All the brilliant people working on the same thing" 8 Jan. 2018, <http://mobprogramming.org/>. Accessed 7 Oct. 2018.

Swarming

“The fiercest serpent may be overcome by a Swarm of ants.” — Isoroku Yamamoto



Auklet flock, Shumagins 1986;
public domain <http://images.fws.gov/>

Unforeseen issues and impediments surface during the project. Impediments are put into the backlog to worry about them later, probably growing the backlog endlessly..

How can we address and resolve unforeseen issues and impediments in an effective and timely way?



Unforeseen issues and impediments will inevitably surface during the project and it might be hard to react in a timely and efficient way

There is various expertise among individuals of the team, however it can be difficult to take advantage and benefit from these experiences.

Sharing knowledge across the team about a solution can be challenging and take considerable time.



Therefore, when a task is critical, have multiple people work together (swarm) on the work item, be it an issue, user story, or impediment. The work can be done by a few or it can be the whole team working together collaboratively to finish the task.

Swarming is a collective behaviour exhibited by entities working together to solve a specific task or move in a specific direction. When dealing with more far-reaching issues like design and architectural questions, swarming provokes a creative dialogue and exchange of ideas without falling into a trap of long and heated arguments. After a while, the team starts working at the same “wavelength” and collective decisions are based on mutual trust. In cases when significant difference in opinion persists longer than usual, the issue can be settled by conducting experiments.

Swarming is usually less involved and less of an intimate relationship between participants than pair programming. A larger group fosters more open discussion and it is much more difficult to impose opinions based on authority and individual personalities. The tacit knowledge and collective intelligence is facilitated by swarming. There is also a more varied mix of expertise and experience as members are free to join and leave the swarm as needed.

These points generally leads to swarming to be less strenuous on the participants. Practical application of swarming has proved that a group environment (as opposed to a pair) overcomes many objections voiced against pair programming (Sargent). A recent study (Laughlin) suggests that groups of three or more are more effective at problem solving than a single person or a pair.

Swarming has been around for quite some time and good agile teams will do what it takes which includes pulling together to do whatever is needed to solve problems and remove impediments. Teams often swarm when there is a serious problem such as system crashes or customer problems. Great teams will use swarming as a planned activity. They recognize that difficult issues will be encountered during development and it is important for everyone to work together and assist each other with resolving problems when they arise.

The following are some advantages of swarming:

- Broader team understanding as the whole team is sharing knowledge about the system;
- Code is better reviewed as the team is constantly reviewing the code;
- "Bus factor²" equals the team member count;
- Continuous learning through sharing of ideas from everyone;
- Efficient problem solving by everyone focusing on single most important task;
- Feeling of fun while working - playful work;
- Less interruptions from email, chat tools or even from other people in person;
- Transparent form of work since all team member know what everybody is doing.

There are also some potential disadvantages to swarming:

- Group's successful will be related to the way its members interacted;
- Groupthink can become part of the swarm, thus not bringing in new ideas;
- Can be inefficient for solving many problems as all people focused on one problem;
- Can have communication overhead as many people interacting;
- Decision making can be slow as swarming looks for consensus.

As swarming has all people working together, there can be challenges where some tasks are too mundane or wasteful to have all participants working together to solve the task. Also, there might be times where certain impediments are blocking the task which might require only 1-2 people a short amount of time to complete. In these situations, *Branching Out* allows for these tasks to be more efficiently addressed by a swarming team. Additionally there can be a task that requires repetitive work throughout the code or system such as fixing a bug. To solve this the group can figure out how to address the task together and then split up and *Spread Out* to solve the task in parallel. Finally to help bring new ideas into the team and to address the issues with groupthink, it is good to occasionally *Keep One in the Dark* by having a member leave the team for awhile or switch with another team and then come back with fresh ideas and can also look at and challenge the groups thinking.

² **Bus factor**, or truck or lottery factor is the number of team members that have to be run over by a bus (or win a lottery) in order for some knowledge to be lost (https://en.wikipedia.org/wiki/Bus_factor)

Branch Out

“You must accept that you might fail; then, if you do your best and still don't win, at least you can be satisfied that you've tried. If you don't accept failure as a possibility, you don't set high goals, you don't branch out, you don't try - you don't take the risk.” — Rosalynn Carter



The team is swarming while adding features. During the development, an impediment appears, preventing the team from collectively bringing items to a close.

How can a team deal with emerging impediments efficiently while swarming?



Keeping the team together with everyone working as a whole is a core principle of swarming or mobbing but implies handling with a single work item at a time.

Some tasks can seem not demanding enough and look like a waste of time for the whole team to focus on each one at once



Therefore, branch out by having a team member or subset of the team break out to work on the emerging issue.

There are times when there are issues or impediments where swarming might not work best. This is a time to turn on the "parallel" mode and have a member, pair or a group break out from the team and work on removing the impediment. After the impediment is removed, those who branched out return to the main mob. The most important difference in the “parallel” mode approach compared to swarming or mob programming team setup is the fact that while the team as a whole is focused on bringing a single work item to a close, there still might be a focus individually, in pairs or in smaller groups to remove different impediments that stand in the way of completing the current work item.

Another situation when branching out is warranted is when some routine, low-complexity work needs to be performed. Such work presents only a small opportunity for learning. Performing it in unison does not provide interesting benefits when all team members are already familiarized with the procedure and automation is not warranted. In those situations a member “branches out” to perform the work and joins the rest of the team once this “solitary” work is done.

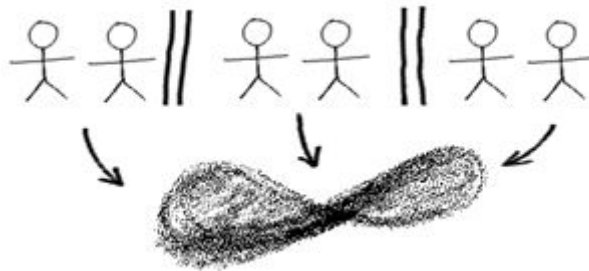
A similar approach would take place when some urgent issue is reported: one or two members would branch out to investigate and, if possible, resolve the issue. In the meantime, the rest of the swarm would carry on with original work.

One situation when the swarm should avoid the urge to branch out is when an automated solution is viable. In this case, it is better to continue as a mob and work on automation of a current problem, than to branch out to find a quick fix.

Spread Out

“Spread love everywhere you go. Let no one ever come to you without leaving happier.”

— Mother Teresa



We are swarming working on tasks and we found a defect or change that needs to be uniformly applied across the code base.

How can we apply a crosscutting change across a large code base efficiently and effectively?



Some tasks require a lot of work across the code base possibly requiring many team members for a considerable amount of time.

To use all team's knowledge is always useful to effectively and efficiently design and implement the change to fix an issue or defect.

There can be some repetition to apply some change or fixing something.



Therefore, after resolving the first occurrence as a group, have the team “spread out” to apply the solution to the rest of the system.

The team can either do this individually or might break up into pairs to apply the solution. After the work is done, the team would rejoin to comment on different facets of the defect and how these were resolved, as well as to document the experience. They can also verify that the changes were applied correctly. *Spread Out* is similar to *Branch Out*, however, spread out implies putting parallelism to maximum use. Branching out implies a small group dedicated to resolving tangent issue. You would only *Spread Out* if there are a lot of tasks that can be done in parallel.

Once again the swarm should avoid the urge to *Spread Out* when an automated solution is viable, In this case, it is better to continue swarming and work on automation of a current problem rather than splitting the team up. It might make sense to have a pair *Branch Out* to experiment with a spike solution to see if automation is a viable alternative. In this case, if there is a viable solution to automation, these ideas should be brought back into the swarm for final integration and validation.

Keep One in the Dark

“Walking with a friend in the dark is better than walking alone in the light.” — Helen Keller



We are swarming evolving the system applying the best practices. However the teams seems to have lost the creativity and everyone seems to have the same ideas.

How can we continue to evolve, grow and learn as a team?



Uniformity can stifle creativity and lead to biased decisions, motivated by conformity, a phenomenon known as groupthink (Janis).

Diversity in a team’s knowledge and skills helps to generate options for solving problems but may lead to long debates and decision-making processes.



Therefore, occasionally have one person abandon the swarm until a certain task is finished. This can include having another person join the swarm from outside the team to bring new ideas into the team.

After completing the task, a code review would be performed where the solution was presented to the team member temporarily excluded from the swarm and where they are able to question the decisions taken. Their suggestions could then be incorporated into the solution if warranted.

Another possibility when there are multiple teams it to swap team members on a regular basis. This can help bring new ideas and to reduce groupthink.

Summary

Swarming is the act of two or more team members coming together to work on tasks together. Mobbing is a form of swarming where the whole team works together in a fast succession on a shared workstation. Various swarming techniques has been successful in various environments and can be a catalyst for a profound change in team structure and organization. Swarming usually requires intense collaboration, coupled with mutual respect and autonomy harnessed collective intelligence. When properly done, teams have had results of continuous learning and reach new levels of productivity [woody's experience report paper].

This paper has presented patterns of behavior that address issues that arise when swarming. For example, how does one split and merge when appropriate (Branch out and Spread Out). Additionally by Keeping One in the Dark, a team can counteract groupthink and continue to grow and challenge itself.

Acknowledgements

We'd like to thank our shepherd Yann-Gaël Guéhéneuc for his valuable comments and feedback during the SugarLoaf PLoP 2018 shepherding process. Finally we'd like to thank our 2018 SugarLoaf PLoP Writers Workshop Group, XXX, YYY, and ZZZ for their valuable comments and suggestions.

References *****NOTE TO SHEPHERD THESE WILL BE UPDATED*****

- [DeMarco] DeMarco T., *Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency*. New York: Broadway Books a division of Random House, 2002.
- [Kahneman] Kahneman, D., *Thinking, Fast and Slow*. New York: Farrar, Straus and Giroux, 2013.
- [YWA] Yoder J., Wirfs-Brock R., and Aguiar A., "QA to AQ: Patterns about transitioning from Quality Assurance to Agile Quality," 3rd Asian Conference on Patterns of Programming Languages (AsianPLoP), Tokyo, Japan, 2014.
- [YW] Yoder J. and Wirfs-Brock R., "QA to AQ Part Two: Shifting from Quality Assurance to Agile Quality," 21st Conference on Patterns of Programming Language (PLoP 2014), Monticello, Illinois, USA, 2014.
- [YWW] Yoder J., Wirfs-Brock R. and Washizaki H., "QA to AQ Part Three: Shifting from Quality Assurance to Agile Quality: Tearing Down the Walls," 10th Latin American Conference on Patterns of Programming Language (SugarLoafPLoP 2014), Ilha Bela, São Paulo, Brazil, 2014.