Learning to Experiments, Set Thresholds, Hold Measurement: Three more patterns in a Software Analytics Pattern Language

JOELMA CHOMA, National Institute for Space Research - INPE
EDUARDO MARTINS GUERRA, National Institute for Space Research - INPE
TIAGO SILVA DA SILVA, Federal University of São Paulo - UNIFESP

Software analytics is a data-driven approach to decision making, which allows software practitioners to leverage valuable insights from data about software in order to achieve higher development process productivity, and improve many aspects of the software quality. Although widely adopted by large companies, software analytics has not yet reached its full potential for broad industrial adoption. Usually, software practitioners do not use analytics of data generated during the software development process to inform their decisions. Decisions based on practitioners' feeling and intuition can lead to wasted resources and increase the cost of building and maintaining the software. This paper introduces three patterns focusing on how to incorporate software analytics into agile practices on a continuous basis in order to inform the decision-making process of the software practitioners. These patterns are part of a pattern language that intends to present recurrent solutions in software analytics area.

Categories and Subject Descriptors: D.2.8 [**Software and its engineering**]: Software creation and management—*Metrics*

General Terms: Software Analytics

Additional Key Words and Phrases: Software Analytics, Decision Making, Agile Software Development, Patterns, Software Measurement, Development Teams.

ACM Reference Format:

Choma, J. Guerra, E.M. and Silva, T.S. 2018. Towards to Patterns Language for Implementing Software Analytics in Development Teams. jn 2, 3, Article 1 (May 2010), 9 pages.

1. INTRODUCTION

Nowadays, being able to gain insightful information from data and use it in decision making is becoming more and more important. Increasingly companies around the world seek information on data to make decisions about their business. Most companies adopt processes and analytical methods to become more competitive. Data analytics plays a major role in this context, and has been widely used in marketing to achieve and better understand customer behavior and their consumption patterns. The concept of analytics is related to the use of analysis, data, and systematic reasoning to make better decisions [Davenport 2009].

More recently, researchers and professionals in the software development area have been using analytics on data generated during the software development process [Buse and Zimmermann 2010]. By software data, we mean the data generated from source code, bug reports, and test executions recorded in software repositories such as version control systems and issue-tracking systems, as well as the information about usage data typically

This work is supported by the Widget Corporation Grant #312-001.
Author's address: D. Pineo, Kingsbury Hall, 33 Academic Way, Durham, N.H. 03824; email: dspineo@comcast.net; Colin Ware, Jere A. Chase Ocean Engineering Lab, 24 Colovos Road, Durham, NH 03824; email: cware@ccom.unh.edu

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 17th Conference on Pattern Languages of Programs (PLoP). PLoP'10, October 16-18, Reno, Nevada, USA. Copyright 2010 is held by the author(s). ACM 978-1-4503-0107-7

stored in the log files. Within this context, Software analytics (SA) emerged as a data-driven approach to decision making, which allows software practitioners to leverage valuable insights from software data to improve their process and many aspects of software quality.

Software analytics involves monitoring, analysis, and understanding of data extracted from the software development context. This approach is focused on getting insightful and actionable insight for informing the decision making. Insightful information refers to accurate and in-depth information, while actionable information refers to information with real practical value [Zhang et al. 2011].

Although widely adopted by large companies, software analytics has not yet reached its full potential for broad industrial adoption. For small companies, software analytics is an open question and rarely addressed [Robbes et al. 2013]. In a software development context, many decisions related to a software system, such as allocation of development and tests resources can be based on software data analysis. However, software practitioners – owners, maintainers and developers – tend to make many daily decisions based on their experience, feelings, and intuitions – e.g. determining which parts of software need increase test coverage, or which parts of software should be re-factored. This lack of decisions that are evidence-based, that is, any strategy that is not derived from or informed by software data and metrics can lead to waste of resources and an increase in the cost of building and maintaining the software [Hassan and Xie 2010].

Therefore, making good use of software analytics can help agile teams to drive better their decisions and save efforts in the course of the project. To date, however, there is no consolidated approach on how to introduce software analytics concepts and practices into an agile environment aimed at improvements to the development process and software quality.

Seeking to fill this gap, we have identified a set of patterns based on experiences reports in the software analytics area. As previously published in [Choma et al. 2017], our patterns focus on how to incorporate software analytics into agile practices on a continuous basis in order to inform the decision-making process of the software practitioners. In this paper we expand on ways for implementing software analytics in development teams by writing three additional patterns: LEARNING TO EXPERIMENTS, SET THRESHOLDS, and HOLD MEASUREMENT.

We emphasize that these patterns can address different types of concerns. For example, such issues may be related to the source code (e.g., code quality, bug proneness, number of defects, and amount of effort to fix bugs); development process (e.g., productivity and ROI); product business (e.g., usage of features, data quality and user satisfaction); and software runtime properties (e.g., performance, number of transactions and error log).

An overview of the SA patterns showing how they relate to each other is depicted in Figure 1. The blocks in gray represent the five patterns documented in previous study [Choma et al. 2017], while the ones in black are documented in this paper. The blocks in white represent the expected outputs from the application of the patterns. Questions included among the patterns refer to factor that motivates the application of the pattern. Each pattern represents a step recommended for implementation of *software analytics*. A brief description of each pattern is presented in Appendix A.

2. LEARNING TO EXPERIMENT

Also known as Achieve Experiments, Test a Hypothesis, Learn from Experiments, Run Experiments

The issues that emerge in software analytics can be related to different levels of needs. At the development process level, the team may need to evaluate for instance new methods, tools, or practices. At the product level, the team may need to evaluate the requirements, features, or usage data. At the user experience level, they may need to evaluate product usability, user satisfaction, design aspects, etc. Some of these issues can be investigated through data collected from the development environment and software artifacts such as source code, bug reports, test cases, usage logs, documentation, etc. For other issues, however, the team may need to evaluate the usability of a feature that has not yet been implemented. Or even, the team may have implemented a feature that needs to

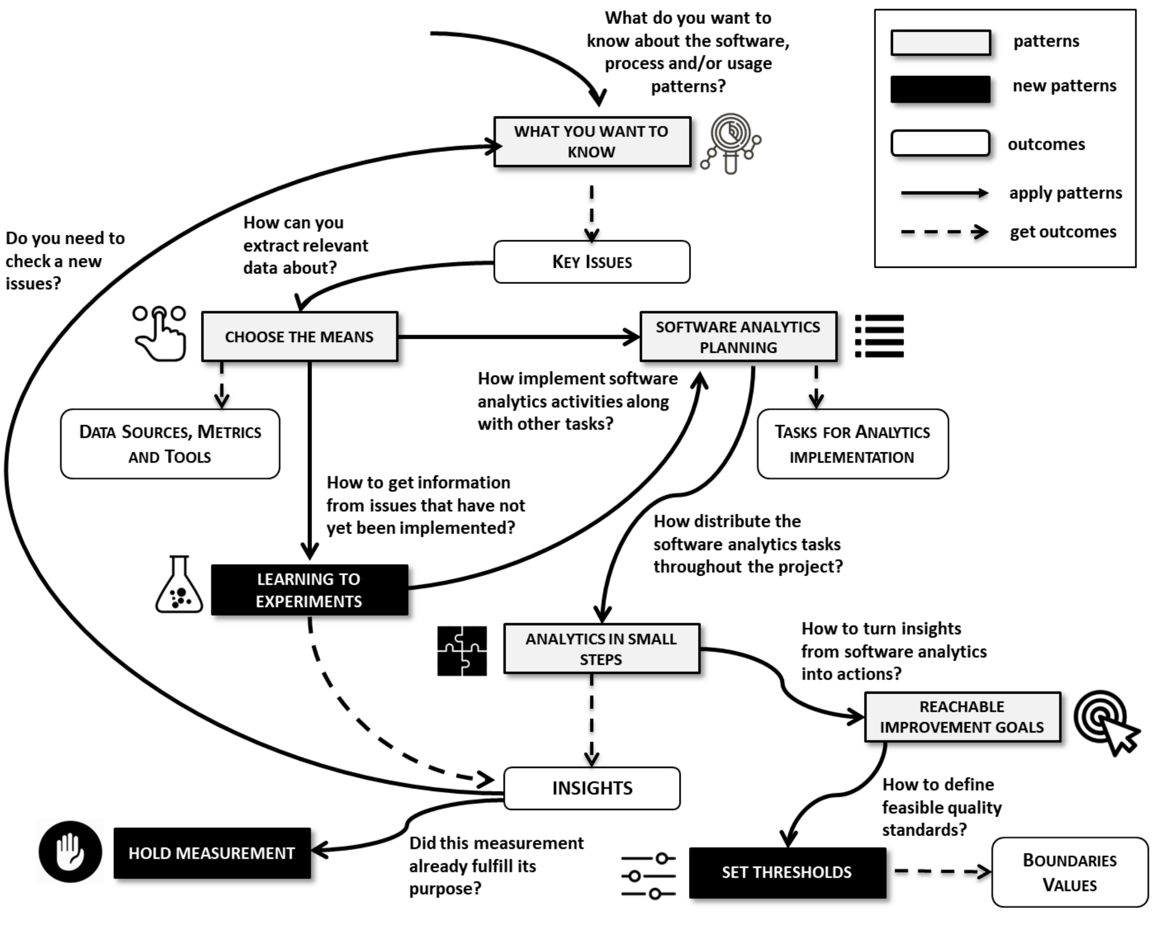

Fig. 1. Overview of the patterns and their relationships.

be improved but does not know how to improve it. In both these situations, the team has nowhere yet to collect and analyze data in order to support their decisions.

How can we obtain information to make informed decisions about software issues on some aspect we haven't yet implemented or we need to redesign?

—The team often has more than one way of implementing the same feature, using different methods and tools for development, and adopting different alternatives of architectural design, but they need to seek to make their choices as successful as possible.

—Sometimes the team makes a decision that will be simple to reverse if it does not work out, but there are solutions that are worth experimenting before implementation to avoid wasting resources and rework.

—Experiments can fail, but learning from both failure and success is important.

—Sometimes, experiments can produce inconsistent results, but the team should investigate the cause of such inconsistencies and then conduct new experiments if they are feasible.

Therefore:

Perform experiments in order to obtain the most insightful information concerning different possibilities and solutions in the software development context from hypothesis formulation.

Experiments can have a low cost of implementation, and their results can provide relevant information to finding the best alternatives for design, tools, approaches to development, test methods, etc. Experiments allow us to test assumptions in order to make the better decisions. However, before opting for experimentation, the team always needs to weigh the cost of doing one or more experiments with the cost of re-engineering or redesigning after you implement something.

The main goal of experimentation is to verify whether an idea is promising or it makes no sense to continue with it. The team needs to have a clear purpose for the experiment and have a good hypothesis to test.

The experimental design should be carefully planned and the experimenters should clearly know which aspects of the software or process will be observed. To ensure successful experimentation, the results should be analyzed without bias by the development team.

During the experimental design planning, the team should be especially careful to also define the experiment size. Large experiments can be costly and unfeasible. Both ROI and the time to implement an experiment are important factors to be considered by the team before adopting them.

Sometimes experiments can fail and sometimes they can produce conflicting or unclear results. When an experiment fails or has unclear results, the team decides if new experiments should be carried out from the lessons learned. Replicating an experiment may be impractical depending on its cost and size. Small experiments tend to be cheaper and easier to replicate. With regard to experimenting and learning, worth taking into consideration Linda Rising's advice:

> *"You can't realistically plan anything from the beginning; the only way to reach your long-term goals or solve your big problems is to try a small thing and learn from the experience. That's how we have always learned. Babies do this from the start. It's the basis for the scientific approach. Experiment and learn."* [Rising 2011]

As an example, imagine that a development team wants to increase the number of hits/clicks on related products in an e-commerce application. Currently, in this application, the products are simply recommended according to the category. The team has the following idea: an algorithm to recommend products that were recently bought with the product being searched. Additionally, the team wants to change position the related products in the user interface. They do not know how much it will be pleasing to the end-user. To save development effort, the team develops some prototypes and performs an experiment with a limited number of users, representing the target audience. From the experiment results, they were able to know which was the best option for the user interface redesign.

As a consequence, the experimentation results can produce insightful information about the product or the development process, that is, the reliable and valuable knowledge needed to make better decisions.

Sometimes, team members can be biased in how they interpret results of an experiment. If it doesn't produce the results they expect, they may discount the results or find ways to invalidate the experiment. Other times, the results of an experiment may be inconclusive. In that case the team must decide whether to perform another experiment of to pick among equally viable options.

Sometimes, team members can be biased in how they interpret the results of an experiment. If it doesn't produce the results they expect, they may discount the results or find ways to invalidate the experiment. Other times, the results of an experiment may be inconclusive. In that case, the team must decide whether to perform another experiment to pick among equally viable options. Also, an experiment may fail. In that case, the team may have to spend time figuring out why something you thought would improve the system did not.

◇ ◇ ◇

The interaction patterns in user interfaces proposed by Welie and Trætteberg [2000] are focused on solutions to problems end-users have when interacting with systems. These patterns can help to analyze the results of the experiment by identifying usability issues essential to interaction design quality. Futhermore, Perzel and Kane [1999] and Montero et al. [2003] proposed a set of patterns focused on website design that can be used in experiments as a support to evaluate user interfaces under usability criteria and facilitate communication between stakeholders and end-users.

3. SET THRESHOLDS

Also known as Define Quality Boundaries, Maintain Quality Level, Follow a Quality Standard, Establish Thresholds

From data collected about an issue addressed by software analytics, the team analyzes their findings and discusses possible solutions and insights to make better decisions. From their insights, the team defines what goals they want to achieve with respect to emerging issues, considering the improvements that can be made incrementally. After implementing the improvements via informed decision making, the team can evaluate the impact of the changes by collecting feedback from stakeholders. Once the goals have been achieved, the challenge will be to maintain the quality level achieved.

How can you maintain a quality level for important aspects of the software?

—By analyzing software data, developers can make better decisions about improving the development process and software quality, but the quality of some aspects will need to be continuously inspected.
—The culture of continuous improvement is stimulated by achieved goals and satisfaction of stakeholders, but it may not be easy to convince stakeholders about the tradeoffs of continuous inspection.
—The process of continuous improvement helps sustain the software evolution and maintenance, the team must have actionable goals and establish quality standards.

Therefore:

Establish the quality thresholds for any issue that the team intends to keep continuously inspecting.

By continuing to collect data, the team will have enough information to decide whether to consider an issue to be resolved, or whether the issue should be monitored for longer. With respect to unresolved issues, the team will need to decide whether these issues will be re-analyzed using new data, or put on hold. Ideally, the team should establish the thresholds values for any issue that the team decides to evaluate or to keep in monitoring. For example, for issues related to coverage testing, the response time cannot exceed 2s or the test coverage must be at least 70%. In general, the team must establish its quality thresholds whenever there is a need for continuous inspection, and redefine them when necessary. The acceptable minimal values for system quality attributes should be periodically analyzed and redefined focusing on continuous improvement. Moreover, the team may need to make tradeoffs between different software aspects – e.g., performance, security and usability. Thus, the threshold value of an aspect can be redefined so that another aspect can work.

As an example, supposing that the team wants to automate more of their tests, but they do not know where to start, once the software has an immense amount of classes. Their key issue is "Where should we focus our test efforts?". To answer this question, they identified the need to investigate two data sources: the code-source to verify current test coverage, and the code repository to verify the percentage of commits related to fixing bugs and the classes with the highest number of changes in order to identify the most "problematic" classes. As data gathering mechanisms they decided (a) to adopt SonarQube for code coverage; (b) to find a tool to collect the number of changes, and (c) to develop a script to relate commit messages with bug issues. When analyzing the collected data looking for insights, the team found that "Web controllers have a high change rate and a low coverage" and "many changes in DAOs were related to bug fixes". Then, as incremental goals, they established a minimum class coverage for Web Controllers of 60%; and a minimum class coverage for DAOs of 80%. Going forward for new classes, they SET THRESHOLDS on at least 80% coverage.

As a consequence, as new requirements come in, the team is engaged in evolving the software while maintaining a quality standard. By establishing these boundary values, the team assumes a commitment to maintaining software quality.

As a consequence of focusing on meeting any threshold, the team may focus on it and ignore other issues.

◇ ◇ ◇

The SYSTEM QUALITY DASHBOARDS pattern [Yoder and Wirfs-Brock 2014] recommends the use of dashboards to monitor important qualities aspects from values established by the team. Tools for monitoring systems such as SonarCube allow you to configure alerts and notifications when measured values cross a threshold. The CONTINUOUS INSPECTION pattern [Merson et al. 2014] captures the overall practice of continuous inspection to preserve the quality of the source code and its alignment to the architecture in an agile environment.

4. HOLD MEASUREMENT

Also known as Suspend Measurement, Discard This Measurement, Drop this Measurement, Break Measurement

When investigating a potential problem related to software building and maintaining, the team first performs data gathering and then analyzes the data to find relevant information. If there are significant findings, the team can propose REACHABLE IMPROVEMENT GOALS to solve the problems found. Sometimes, it is necessary some effort to automate or continuously monitor this data gathering and analysis. Specially when the information obtained was enough for the answers needed or the problems have a low possibility of recurrence, it might have a low priority for the team.

What should the team do when an implemented measurement is not a priority in the current project and takes some effort to be continuously monitored?

—It is critical that the development team make decisions in its process based on data about the software, but these activities need to be carefully planned because they can consume a lot of team effort.

—Emerging issues need immediate investigation to avoid software operation failures and information inconsistency, but the team has other priorities in the project.

—A given issue that has been the subject of a measurement may have achieved an important objective, but for the next interactions, it may be unnecessary to continue measuring it.

—Once a metric has been obtained through scripts, such as a static log analysis or a SQL query, it can be costly to add it into a continuous monitoring mechanism or to execute it frequently. The integration of this measurement in the deployment or build environment collecting live data might demand a considerable effort.

—There are tools that can facilitate the process of automating data gathering reducing significantly the effort to make it continuously monitored.

Therefore:

Put on hold the measurements that already fulfilled their initial goal, demand effort to be continuously monitored and are not considered currently a priority by the team.

Faced with problems related to the software usage, the team may need to check specific issues. The team is suspicious of some flaws in the system, but they have no idea about the dimension of the problem, nor about the real impact upon system operation. They decide to investigate the issue through further analysis. In one-off action, they detect the problem through software analytics. Then, they define the next steps to solve the problem and put the process used to collect and analyze information on hold. The team can suspend measurement of the issues with a low possibility of recurrence. However, some issues may need monitoring for a longer period in order to avoid flaws recurring in system operation. At that moment, however, the team has defined that, for some reason (e.g., effort, cost or other project constraints), the monitoring of these issues cannot be implemented immediately.

When investigating and detecting potential problems, the team can establish metrics to continuously monitor them. In many cases, the team may not yet have a monitoring system. Or it could be that the current system is overloaded monitoring other issues that are more important, or the monitoring system does not provide resources to monitor a particular type of problem.

The cost of collecting data either manually or in a one-off way may be less than implementing a continuous monitoring solution. For instance, consider a data extracted from the database using an SQL query or information extracted from a log analysis using a simple script. It demands effort to create a feature that continuously extract that information and provide it to the team. So, the team needs to assess when it is feasible or not to continuously monitor that metric taking into account their priorities and the cost of implementation.

As a practical example of when to HOLD MEASUREMENT, suppose that a team is using software analytics to know about the consistency of the information stored in the database where data are provided daily (minute by minute) from a distributed sensors network. The team suspects data inconsistency caused by sensor failures, but they do not know the extent of the problem. From the analysis of data, the team has obtained evidence to prove their suspicions and make some decisions about what to do to resolve this problem. In contact with the domain experts, they discuss mechanisms to normalize the data before the information is delivered to the end user of the application. They envisage the possibility of implementing a continuous monitoring on the issue, but currently they have other higher priorities and demands. Because of this, they decide to put the measurements on hold. As an advantage, after this experience, the team already has the knowledge on how to collect and analyze sensor data any moment they need to in the future.

As a consequence, through leaner methods, the team acts quickly to gather evidence on issues of concern, which can have an irreparable effect if they are ignored for a long time. Thus, more sophisticated solutions to the measurement of the problems can be best planned to occur at the most appropriate time. Moreover, the team can perform this process once more when necessary to verify the same type of problem.

Sometimes, the team may postpone the continuous monitoring of essential metrics for maintaining the proper functioning of the application.

◇ ◇ ◇

The RECALIBRATE THE LANDING ZONE pattern [Yoder and Wirfs-Brock 2014] is related to this pattern by addressing the implementation of decisions when resources are incrementally implemented. It is natural that criteria adopted in the course of the project need to be adjusted over time. These decisions can affect or limit the ability to achieve new goals and meet other demands. Thus, measurements may be provisionally suspended and then refined in future actions. The ARCHITECTURAL TRIGGER pattern [Wirfs-Brock et al. 2015] suggests that when the team does not know when to evolve the architecture they can develop architectural triggers. Similarly, the team can define triggers to warn them when a certain condition may require immediate action to treat a certain issue.

5. SUMMARY

In this paper, we have presented three more patterns in a Software Analytics pattern language. These patterns are intended for software practitioners — including project managers, analysts and software developers from small, large, or multiple teams. Taking into account the different levels of decision-making, the proposed patterns describe steps to integrate analytics activities into the development process.

Appendix A: Summary of Patterns Collection.

Next, we present a summary of the eight patterns containing a brief description of each of them.

(1) WHAT YOU WANT TO KNOW: To solve how to drive the process of selecting and collecting metrics so that they are useful to the team for decision making, in a context where there is a large amount of software data that can inform the decisions of the team, the solution is to define the key issues that the development team wants to focus on, in order to guide their selection of the appropriate means for measurement, assessment and monitoring these issues throughout the project.

(2) CHOOSE THE MEANS: To solve how to gather data about the issues that you intend to answer during the project, in a context where there are various possibilities of choice, the solution is to define the data sources and most appropriate means, such as tools, techniques and other approaches for selecting and collecting data that will be useful for future decisions.

(3) SOFTWARE ANALYTICS PLANNING: To solve how the tasks of *software analytics* should be implemented along with the other tasks, and fitted appropriately into the project planning, in the context where the tasks directly related to the implementation of software features has high priority, the solution is add tasks related to the *software analytics* on the to-do list to be prioritized with the regular project tasks according to the team's demand for information.

(4) ANALYTICS IN SMALL STEPS: To solve how to implement *software analytics* at a pace that it does not impact project activities and provide enough information for decision making, in the context where much information at the same time can confuse and make the team lose focus, the solution is to distribute tasks related to the *software analytics* throughout the project, adding information to the team about the system at small portions by adjusting the granularity of the analytic activities.

(5) REACHABLE IMPROVEMENT GOALS: To solve how to turn insights from *software analytics* into actions to incrementally improve the characteristics of the software system in a short time, in a context where to perform all improvements based on the analytics automated feedback might lead the team to act without focus, the solution is define reachable improvement goals from the *software analytics* findings, and break the activities down into smaller tasks to fit together with the other tasks.

(6) HOLD MEASUREMENT: To solve if an issue still need to be continually monitored after some initial measurements, in a context where the team does not yet have a monitoring system, or the current system is overloaded with other issues, the solution is to suspend measurement of the issues with a low possibility of recurrence, or of the issues that need to be continually monitored but the team has defined that, for some reason (e.g., effort, cost or other project constraints), the monitoring of these issues cannot be implemented immediately

(7) LEARNING TO EXPERIMENTS: To solve how to obtain knowledge about the product from real data to inform our decisions, in a context where the team needs to develop new features not yet implemented, or even improve ones already deployed, the solution is to perform experiments involving customers and real users in order to obtain the insightful information needed to guide the development decisions of the software features.

(8) SET THRESHOLDS: To solve how to maintain a quality level of important aspects of the software, in the context where the improvements can be made incrementally, the solution is to establish the quality thresholds for any issue that the team intends to keep in continuous inspection.

REFERENCES

BUSE, R. P. AND ZIMMERMANN, T. 2010. Analytics for software development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 77–80.

CHOMA, J., GUERRA, E. M., AND SILVA, T. S. 2017. Patterns for implementing software analytics in development teams. In *Proceedings of the 24th Conference on Pattern Languages of Programs*. The Hillside Group, 12.

DAVENPORT, T. H. 2009. Make better decisions. *Harvard business review 87,* 11, 117–123.

HASSAN, A. E. AND XIE, T. 2010. Software intelligence: the future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 161–166.

MERSON, P., AGUIAR, A., GUERRA, E., AND YODER, J. 2014. Continuous inspection: a pattern for keeping your code healthy and aligned to the architecture. In *3rd Asian Conference on Pattern Languages of Programs, Tokyo, Japan*. 6–8.

MONTERO, F., LOZANO, M., GONZÁLEZ, P., AND RAMOS, I. 2003. A first approach to design web sites by using patterns. In *First Nordic conference on Pattern Languages of Programs: VikingPLoP*. 137–158.

PERZEL, K. AND KANE, D. 1999. Usability patterns for applications on the world wide web. In *Proceedings of the Pattern Languages of Programming Conference*. Vol. 99. Citeseer.

RISING, L. 2011. Small experiments. *Better Software* Jan-Feb, 13–44.

ROBBES, R., VIDAL, R., AND BASTARRICA, M. C. 2013. Are software analytics efforts worthwhile for small companies? the case of amisoft. *IEEE software 30,* 5.

VAN WELIE, M. AND TRÆTTEBERG, H. 2000. Interaction patterns in user interfaces. In *7th. Pattern Languages of Programs Conference*. 13–16.

WIRFS-BROCK, R., YODER, J., AND GUERRA, E. 2015. Patterns to develop and evolve architecture during an agile software project. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*. The Hillside Group, 9.

YODER, J. AND WIRFS-BROCK, R. 2014. Qa to aq part two: Shifting from quality assurance to agile quality. In *21st Conference on Patterns of Programming Language (PLoP 2014), Monticello, Illinois, USA*.

ZHANG, D., DANG, Y., LOU, J.-G., HAN, S., ZHANG, H., AND XIE, T. 2011. Software analytics as a learning case in practice: Approaches and experiences. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*. ACM, 55–58.

Received February 2009; revised July 2009; accepted October 2009

PLoP'10, October 16-18, Reno, Nevada, USA. Copyright 2010 is held by the author(s). ACM 978-1-4503-0107-7