Framework for Describing and Analyzing Context and Factors for Software Engineering Research: Applying the SEMAT Kernel

Pan-Wei Ng

Abstract—There is currently no widely accepted approach to report software engineering research findings. It is not easy to understand the context from which a research work is conducted and factors that influences the results, and hence difficult to evaluate the applicability of findings to another context. This paper proposes a framework for describing and analyzing context and factors for software engineering research. Our proposed framework is an approach towards Theory Based Software Engineering (TBSE) that relates contextual factors and process improvement objectives systematically. Our approach towards TBSE is centered on a software engineering kernel known as Essence. Essence originates from the recent global SEMAT (Software Engineering Method and Theory) initiative and is currently an emerging standard with OMG (Object Management Group). Essence helps identify architecture views to describe the context of a software engineering endeavor. It also helps identify factors of success and make recommendations precise and actionable. This paper demonstrates the application of our proposed approach on a telecommunications case study. This case study by itself is interesting because it yielded 21% productivity gains and 58% decrease in defects. But more importantly, it demonstrates the viability of our approach in making theory practical to development teams in a systematic manner.

Index Terms—Software engineering, empirical, SEMAT, theory, essence, alpha, architecture view, process improvement, success factors, theory, telecommunications.

I. INTRODUCTION

Many results and tools from software engineering research often failed to the industry [1]. Industry needs more than just mere explanation of techniques and approach. It also needs empirical evidence to demonstrate and justify their use. Today, we see growing but nevertheless limited interest in empirical software engineering research [2], [3]. However, a systematic framework for reporting case studies and findings is still lacking. As a consequence, it is difficult to search for relevant case studies, compare findings and generalize results. Dyba^o and Dingsøyr [4] surveyed research for empirical evidence of agile software development and found that different reporting content hinders analysis. Jedlitschka, Ciolkowski, and Pfahl [5] also pointed out that a major problem for integrating software engineering research results into a common body of knowledge is the diversity of reporting styles. It is difficult to locate relevant information; and important information is often missing. Petersen and

Wohlin [6] found that studies investigating a similar object do not agree on which context facets are important to mention. Thus, there is a clear need in academia for a systematic framework for reporting findings.

A. Framework for Software Engineering Research

Before proceeding further, we would like to clarify what we mean by a framework. We recognize that there have been generally accepted approaches on conducting research and reporting case studies, such as proposed by Runeson and Höst [7]. However, these focus more on research methods. We, on the other hand, want to a framework to help organize research data, i.e., the factors that influences the results of software engineering research. Examples of factors include team size, team distribution, system complexity, and stakeholder diversity. Framework in this paper means a domain model for software engineering.

The key problem is that software development is complex whose success depends on a large number of factors. Boehm's COCOMO [8] has a basket of 15 factors. Clarke and Connors [9] reviewed a number of literatures to make the number of factors more comprehensive resulting in 8 classifications, 44 factors and 170 sub-factors. Chow and Cao [10] found 36 factors affecting the success of agile development. The large number of factors poses serious challenges to practitioners:

- Classification – Is there a way to organize these factors? How to find them? Are two factors the same (overlap)?
- Context – As Dyba [11] and Kruchten [11] pointed out, practitioners need to evaluate which factors are more important them and understand When during development should certain factors be emphasized or downplayed? When should development teams do something about it, when should they move on?

B. Factors and Theories of Software Engineering

Understanding how factors relate to one another is in fact formulating a “theory” about how things work. Most practitioners are familiar with the theories and laws of physics. For example, the acceleration of a parachute or a falling object is positively influenced by the gravitational pull and negatively by air drag, which is in turned positively influenced by the velocity of the object (see 0). In 0, a directed line denotes an effect from one end to another end, with a circle having a sign to show a positive (contribution) or negative (inhibition) influence.

Experimentation and physical models helps us evaluate the magnitude of each influence. Research and engineering in physical sciences are about finding the magnitude and factors of each influence and finding techniques that change the

Manuscript received February 5, 2014; revised April 10, 2014.

Pan-Wei Ng is with Ivar Jacobson International, Singapore (e-mail: panwei@ivarjacobson.com).

influence. For example, the design of a parachute tries to increase its surface area to arrest more air, whereas a missile is streamlined to reduce drag (see Fig. 1).

Fig. 1. A physical theory that relates factors affecting parachute motion.

Similarly, theories of software engineering relate software development success (e.g. faster time to market, quality, costs, motivation, etc.) and their factors. (e.g. teamwork, getting feedback quickly, having an extensible architecture, getting stakeholders involvement, etc.). Much software engineering research efforts have aimed to identify software development factors and their effect on software development success. Some investigated narrowly scoped relationships such as been code dependencies and defects [12], [13]. Some investigated the use of broader relationships such as between architecture and human organization structures as in the case of Conway's law [14], [15]. The current GTSE (General Theory of Software Engineering) workshop series [16], [17] inaugurated in 2012 has been investigating the formulation of a general theory for software engineering.

C. The SEMAT Initiative

GTSE is part of a larger and global Software Engineering Methods and Theory (SEMAT) [18], [19] initiative founded by Ivar Jacobson, Bertrand Meyer and Richard Sole, aimed precisely to mature the state of software engineering and help teams learn from one another. SEMAT's first step was to identify the universal concepts common to all software engineering endeavors, both traditional and agile. The result is a software engineering kernel known as Essence [20], which is now an emerging standard with the OMG (Object Management Group). Essence provides an extensible and

actionable object-model helps teams detect risks early and achieve progress in their software engineering endeavor. This object model has objects (which are called alphas) such as Stakeholders, Opportunity, Requirements, Software System, Work, Team and Way of Working. On top of this foundation, teams can then select practices specific to their needs. Even by itself, Essence is helpful for development teams. Moreover, the novel use of (poker-sized) cards makes Essence lightweight, fun and engaging [21]. Essence has been applied in both small and large industry projects [22] and also in education [23], [24].

D. Our Proposed Framework: Theory Based Software Engineering

With growing interest, especially in the concept of alphas, Essence is poised as an attractive framework for reporting empirical research and case studies. Essence In particular, we recognize the potential of Essence to provide a domain model to describe software engineering context and to relate the influences of factors on objective measures and hence recommended practices (see Fig. 2).

0 depicts our proposed framework, which we call Theory Based Software Engineering (TBSE). TBSE begins by describing and analyzing the context of a software engineering endeavor. This gives context to measures and factors of interests that are related and explained by some a specific (customized or contextualized) theory. Such specific theories are based from known and more general theories that one can find readily in available literature, such as Boehm's COCOMO [8], Conway's law, Zimmerman and Nagappan's [12] relationship between defects and program dependencies, etc. The specific theories are used to generate recommended practices. Observations made when applying these practices serve to validate and tune the context-specific theories, which in turn validate the general theories. At the same time, these practices will also improve the maturity of the software engineering endeavor.

Fig. 2. A theory based software engineering framework.

Essence plays a key role in describing software engineering context, and in particular laying the foundation for a set of views to describe a software engineering endeavor (such as developing a new software system, enhancing an existing one, offshore development, etc.).

Osterweil noted that software processes are software too [25]. Just as software has architecture, can be measured and analyzed, so do software processes (and methods¹). Just as

software architectures can be described through clearly defined views [26], software methods have architecture views too. For software systems, popular approaches like Kruchten's 4+1 view [27] give developers concrete guidance how to comprehensively describe the context, structure and behavior of software systems. However, no known work exists to discuss architecture descriptions and views of software engineering processes and methods.

There ought to be a set of views for each universal alpha in Essence. Thus, we have views for Stakeholders, Opportunity, Requirements, Software System, Work, Team and Way of

¹ From henceforth, we will use the term processes and methods interchangeably, preferring the word "method" over "process".

Working. For example, the Work view describes the key milestones and events for a software engineering endeavor, i.e. the Software Development Lifecycle (SDLC) as well as regular events on a calendar. The Stakeholder view will show stakeholder relationships and their involvement. These views can be described textually or graphically, such as using UML, depending on the desired level of detail. If a particular view needs greater emphasis graphical depiction is necessary, otherwise, textual descriptions suffice.

E. Objective and Overview of Paper

Our objective of this paper is to demonstrate the use of our Theory Based Software Engineering (TBSE) approach to describe the context of software engineering endeavors and to theorize success factors for process improvement. The context of software engineering endeavors and success factors are organized around the domain model offered by Essence. We demonstrate how to apply TBSE to analyze and explain the success of a telecommunication software product line case study, which achieved 21% productivity gains and 58% decrease in defects. We provide extensive and step-by-step detail to relate to problem and solution factors in this case study. The contribution of these factors had been confirmed through measurements, which are also described in this paper.

We organize the rest of this paper as follows. Section II provides a brief overview of Essence. Section III demonstrates the use of Essence alphas for describing architecture views of our case study. Section IV demonstrates how Essence provides the context for identifies factors affecting our case study, which can then be used to identify factors and practices for process improvement. We further show how results of applying these practices that confirm the identified relationships. Finally, Section V concludes with a brief discussion on ongoing efforts.

II. BRIEF OVERVIEW OF ESSENCE

Fig. 3. Essence Alphas and their states.

In this section, we give a brief overview of Essence. We describe some of the key concepts in Essence [21] namely *alphas* and *alpha states*.

Alphas – Essence uses an object-oriented approach to identify typical dimensions of software engineering

challenges. These objects are called *alphas*. Essence kernel identifies alphas that are common to software development such as *Opportunity*, *Stakeholders*, *Requirements*, *Software System*, *Work*, *Team*, and *Way-of-Working*.

Alpha States – Each alpha has states (see 0) that provide guidance for development teams to achieve progress along these dimensions and to detect risks and problems early. For example, the progress of (a set of) Requirements goes through the following states: *Conceived*, *Bounded*, *Coherent*, *Acceptable*, *Addressed*, *Fulfilled*. Essence kernel provides a detailed checklist for each alpha and their states.

Essence presents the alphas and their states in a lightweight manner using poker size cards. Fig. 3 shows the Requirements alpha card on the left, and Requirement alpha state cards on the right for the *Coherent* and *Acceptable* states. The number at the bottom of each state card denotes its sequence. For example, Coherent is the 3rd out of 6 Requirement alpha states (see Fig. 4).

Fig. 4. Requirements alpha and state cards (coherent and acceptable).

III. DESCRIBING CASE STUDY CONTEXT USING ESSENCE

This section describes how Essence supports TBSE and makes TBSE practical to software development teams. The different alphas in Essence kernel are useful tools for getting a comprehensive understanding of software development endeavor. In practice, it is useful to discuss this with a development team on a big visible Essence Canvas, which is a simple board to make different aspects of software engineering endeavor visible on a whiteboard or a big sheet of paper (see 0). It is the first step for exploring the theories underlying process improvement for a software engineering endeavor. It comprises areas for team members to describe the software development endeavor either using text or diagrams (see Fig. 5).

Fig. 5. Essence canvas.

We will make our application of Essence based TBSE concrete using a case study. This case study occurred in 2012 at a development unit within major telecommunication Product Company. This development unit builds telecommunication transport networks. Each release takes about 9 months. It includes both software and hardware

enhancements and about 80 engineers. Prior to our involvement, they had different teams build different releases in parallel on different branches of the C/C++ source repository. This created additional work in merging both enhancements and bug-fixes. However, managers and developers responsible for individual releases were reluctant to change to working on the same code branch because they were fearful that changes from one release would affect the quality of another release. We describe the case study along the different dimensions (i.e. alphas) as follows:

Opportunity – The development unit is responsible for building a major subsystem in a telecommunication transport network. They deliver business value by delivering features to existing and potential telecommunications operators and reducing product costs (through less expensive hardware). Business value is also about driving down development costs and wastes.

Stakeholders – Marketing represents the needs of end-customers, i.e. telecommunications operators. System engineers who have deep understanding of the transport networks and technologies act as subject matter experts. System engineers also act as the bridge between marketing and development. The development unit's head is responsible for smooth development operations.

Requirements – The requirements for each release are about enhancing features, adding new protocols, upgrading hardware, such as using different chipsets from different hardware vendors. Each release comprised a number of features. Each feature is allocated to a team, led by a team leader. Teams work in parallel.

Software System – The software system, which has in the order of million SLOC, is divided into layers and then into components. Each release of the software system occurs at its own branch.

Team – Each release involves about 80 people include release managers, team leaders, system-engineers, developers and testers. These 80 people are grouped into teams of about 5 to 8 with each team responsible for a set of components in the access network. Development was distributed across three cities but in the same time zone. A system engineer was responsible for analyzing and explaining the feature set to the team from the requirements perspective. The system engineer role was similar to that of business analysts in the IT world (in contrast to the telecommunications world) and usually had no knowledge of the implementation of the impact of his allocated feature set on the software system. Testers joined in development late and did not have the privilege to raise their concerns and queries early during development. Upon completing development, another group of testers will conduct testing from a black box perspective based on their understanding of the requirements.

Work – Each release takes about 9 months end-to-end and usually involves about 200K SLOC. The development teams work in parallel on a 3-week iteration cadence. A release manager is responsible for meeting development schedules. System engineers estimate the development effort based on their experience using SLOC and allocate features to various iterations in the release. They track progress of requirements development work informally using spreadsheets.

Way of Working – Development occurs at two levels –

team level and release level.

- Each development team is responsible for defining, building, and testing their features for a release.
- At regular 3-week cadence, a testing team performs additional functional testing at the release level.

Their way of working was a hybrid combination of traditional silo and iterative development. However, they had a strong tradition of continuous integration. Middle managers including release managers and line managers collaborated very closely. Whenever there was an issue, they would quickly come together and discuss solutions.

IV. THEORIZING PROCESS IMPROVEMENT

In this section, we use Essence to analyze the relationships between software process improvement objectives, recommended actions, which we call solution factors, and their possible negative effects, which we call problem factors. An objectives and theory exploration board (see 0) makes provides a visual aid to achieve this. This is an extension to the Essence Canvas (see 0) by placing an area for objectives at the top. Sticky notes are used to depict objectives in blue, solution factors in green and problem factors in red. Strings make relationships between them visible (see Fig. 6).

Fig. 6. Essence objectives and theory exploration board.

Fig. 7. Mapping problem and solution factors in the new way of working.

We highlight the problem factors (PF) and solution factors (SF) explicit and overlay them on top of alphas and alpha states on an objectives and theory exploration board. 0 shows an influence diagram depicting the relationships between key problem factors and solution factors in our case study. An explosion icon denotes a problem factor (PF). A star icon denotes a solution factor (SF). The directed lines follow the same notation earlier in 0 (see Fig. 7).

PF-1 (Wasteful merging) – The primary objective to address in the case study is to eliminate the non-value-added work to merge changes from one release to another.

SF-1 (Reduce features) – One area of improvement was to relook the Opportunity and to reduce the number of features per release and thereby eliminate the need to have any parallel development. This option was quickly discarded because the product line was aggressively trying to gain market share against competition. Bringing competitive products with more features and at lower costs (through less expensive customized hardware) to the market was an important strategy. These new features, which included hardware upgrade, took time and overlap between releases was inevitable.

SF-2 (Work on the same branch) – The head of development recognized that they had to change the way they implement the system to work on the same branch for different releases.

PF-2 (Tangling, scattering and duplication) – However, the release managers who had the habit of working on different branches were apprehensive. Through repeated addition of features over the releases some part of the implementation had deteriorated very badly. Tangling, scattering and duplication were significant.

SF-3 (Align requirements and code) – Clearly an area of improvement is to improve the way the developers code new features. We suggested that development teams should have better alignment between requirements and code using aspect orientation principles [28].

PF-3 (Late discovery of flaws) – The department highlighted that flaws and defects were usually uncovered during the Requirements Fulfilled State.

SF-4 (Collaborative analysis) – By getting different roles within the team to have an understanding of the acceptance criteria, defects could have been discovered earlier. Thus, the Requirements: *Acceptable* state needed to be strengthened.

PF-4 (Bugs introduced to another release) – As mentioned, the development teams were apprehensive that working on the same branch (SF-2) would introduce bugs from one release to another release and thus creating another problem during the Requirements Fulfilled state.

Our recommendation was that during Requirements Addressed state when the development teams were working on the same branch, they should align requirements and code and thus keep codes between separate features/releases separate (see SF-3 above).

SF-5 (Impact analysis on other features/releases) – To facilitate alignment between requirements and code, it would be important to have a good and well-structured requirements model that shows the dependencies and impact of requirements across features or releases.

PF-5 (Schedule uncertainty) – It is expected that the change towards working on the same branch and towards the above areas of improvements will not happen overnight. Thus, a release would be affected by other ongoing releases and there would be uncertainty due to known code changes from other releases.

SF-6 – To address PF-5, a release that could impact another release should make its impact known to the impacted release. In this way, the impacted release could actively track the impact and make adjustments accordingly.

A. Recommended Practices

By pinpointing where problem factors and solution factors occur in the alphas and alpha states, it became practical and easy to provide actionable advice to team members what the changes to their development approach were and what they needed to be mindful of. 0 highlights the changes in the way the department develop software for each feature according to the alpha states:

- Requirements: Bounded, Coherent: SF-5 – Analyze impact on other releases early to inform affected releases that their release manager may make necessary adjustments to their plans.
- Requirements: Acceptable: SF-4 – Involve the system engineer, the development team and testers to explore the test cases
- Requirements: Addressed: SF-3 – Align requirements and code to provide better separation of concerns. This utilized a number of design patterns. We also had to take into account how much existing code had deteriorated to provide a strategy on refactoring.
- Work: Prepared: SF-6 – As above, inform impacted releases.

This helped the team easily understand what to do when. Of course, we still had to provide training and coaching. For brevity of this paper, we do not go into the details of the techniques used to put the solution factor into effect.

It is important to note that 0 in effect describes a specific theory of how the telecommunications department can improve their development. It highlights important factors relevant to them overlaid on top of Essence alphas and their states. This is a specific theory for this particular telecommunications department as opposed to that general theory embedded implicitly within Essence. In the course of process improvement teams have to be mindful of both theories in action.

B. Process Improvement Results

Our involvement with the development unit spanned from February 2012 to July 2012. We helped it transitioned to a trunk based development approach that permitted parallel of release 5 (R5) and release 6 (R6) of their products. 0 depicts a schematic showing the overlap between R5 and R6. Each release has three checkpoints, namely: the start of development (coding) and two deliveries denoted by Delivery 1 and 2 (see Fig. 8).

Fig. 8. Checkpoint calendar.

Our involvement began before development started in R6 when we analyzed their problems and recommended solutions (described earlier in this section), and provided custom training. The problem factors and solution factors highlighted in the above sections were not uncovered in one sitting, but rather over many sessions of discussions before development started.

When development started, we coached teams on how to align code and requirements, how to refactor legacy code.

The department measured code complexity using a tool called Infusion [29]. This tool produced an aggregate lower bound complexity measure, which it calls Quality Deficit Index (QDI) [30]. A better component has lower QDI value. As shown in 0 components most affected in R6 showed an improvement code structure.

TABLE I: CODE COMPLEXITY BEFORE AND AFTER PROCESS IMPROVEMENT

Components most affected in Release 5	QDI percentage change
A	-13.35
B	-17.17
C	-9.77
D	-13.72
E	-7.37

0 shows committed SLOC code across iterations. Each iteration was three weeks long, where they brought completed features to the Requirements Addressed state.

0 shows a spike in iteration 2. This highlighted a stressful situation when team leaders and middle managers were still undecided if they wanted R5 and R6 to work on the same branch (SF-2). R5 was about to release their Delivery 1 and they had doubts about quality impact from R6. Thus, developers from R6 did not commit their codes in iteration 1. We raised this issue to the department head. He took leadership and told the teams to persevere. Once the developers knew that there was no turning back, they focused on doing the new way of development well. Thus, a huge amount of code was committed in iteration 2 (see Fig. 9).

Fig. 9. Code churn across iterations.

0 depicts the number of defects discovered by the development teams across iterations. The department had a policy of not exceeding 60 defects at anyone time. The huge number of codes committed in iteration did not introduce huge number of defects, and the teams were operating within quality ranges (see Fig. 10).

Fig. 10. Defects Across Iterations found in R6.

More importantly than the defects found in R6 was the number of defects found by R5 that were caused by R6. This was the problem factor PF-4. In fact there were 4 such defects. The R6 team analyzed these defects in detail and found that most of the problem was due to poor alignment of code and requirements (SF-3).

C. Review Results

During the time when R5 and R6 worked on the same branch:

- 1) R5 fixed 1342 defects. Since these fixes were on the same branch as R6, no further effort was needed to merge into R6, yielding a saving of 30 man-months.
- 2) R5 introduced new functionality amounting to 184K SLOC. Since this is on the same branch as R6, no further effort was needed to merge into R6, yielding a saving of 55 man-months.
- 3) There was a dramatic 58% decrease in the number of defects in R6 compared to earlier releases. This yielded a savings of 68 man-months.

Development of R6 took 9 months with 80 staff on average. The savings above amounted to $30+55+68=153$ man-months, which represented 21% savings.

- 4) Development teams became more motivated about improving their work products (requirements, design, code and tests).

These results validate the theory and relationships identified in 0 and thus provide a clear empirical validation of the proposed specific theory.

V. CONCLUSION AND FUTURE WORKS

To Osterweil [25] “software process (methods and practices) is software too”. Surely, software engineering and software engineering research are software too, and like software needs to be well architected and designed. This is not about designing a piece of research, but designing how research from different parts of the world fit together.

The work on Essence is in reality “engineering” software engineering. It applies timeless design principles such as separation of concerns [31], [32] on software engineering through the notion of alphas. In this paper, we show that alphas and their state progressions provide important context information about the success factors are most relevant at a given point in time in a software development lifecycle along different dimensions. Thus alphas (and in general Essence) is not just a language or vocabulary of software engineering, but also a theoretical framework.

We took yet another dive to demonstrate how Essence can be used to craft a specific theory for a case study. This specific theory highlights the important problem and solution (success) factors that are of interest to our case study. This theory operates in parallel as the generic (practice and project independent) theory captured within the Essence kernel.

It has been said that there is nothing more practical than a good theory [33]. Surely, as we seek to formulate and evolve a general theory of software engineering, we must have the practitioners’ context, interests and concerns at heart and make it practical for them, not only in the way they apply the supposed theory, but also the way they learn the theory. Not only that, the way the theory is formulated and evolved should also be practical. Thus, any work on theory should never be divorced from practice. This paper has demonstrated how theory and practice integrates.

TBSE is about integrating theoretical basis into daily software development. Our approach describes existing context using architecture views of existing methods and processes and analyze existing context in the light of general

theories. These general theories are generic patterns that offer generic solutions. TBSE provides a way to contextualize these solutions (i.e. recommended practices) into specific theories to relate key success measures with known factors. In this way, development teams and stakeholders will operate on top of a sound theoretical and practical foundation. Essence, offers great value by providing a domain model to organize the different factors influencing the success of a software engineering endeavor.

We are now applying TBSE on a broader class of software engineering endeavors and fine tuning the procedures to apply TBSE.

REFERENCES

- [1] R. L. Glass, I. Vessey, and V. Ramesh, "Research in software engineering: An analysis of the literature," *Information and Software technology*, vol. 44, no. 8, pp. 491-506, 2002.
- [2] T. Dyba, B. A. Kitchenham, and M. Jørgensen, "Evidence-based software engineering for practitioners," *IEEE Software* 22, no. 1, PP. 58-65, 2005.
- [3] D. Budgen, G. Hoffnagle, M. Müller, F. Robert, A. Sellami, and S. Tilley, "Empirical software engineering: A roadmap report from a workshop held at STEP 2002, Montreal, October 2002," in *Proc. 10th International Workshop on Software Technology and Engineering Practice*, (STEP 2002), IEEE, 2002, pp. 180-184.
- [4] T. Dyba and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Information and Software Technology*, vol. 50, no. 9, pp 833-859, 2008.
- [5] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," *Guide to Advanced Empirical Software Engineering*, pp. 201-228, 2008.
- [6] K. Petersen and C. Wohlin, "Context in industrial software engineering research," in *Proc. the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, IEEE Computer Society, 2009, pp. 401-404.
- [7] H. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131-164, 2009.
- [8] B. Barry, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: COCOMO 2.0," *Annals of Software Engineering*, vol. 1, no. 1, pp. 57-94, 1995.
- [9] C. Paul and R. V. O'Connor, "The situational factors that affect the software development process: Towards a comprehensive reference framework," *Information and Software Technology*, vol. 54, no. 5, pp. 433-447, 2012.
- [10] T. Chow, and D. B. Cao, "A survey study of critical success factors in agile software projects," *Journal of Systems and Software*, vol. 81, no. 6, pp. 961-971, 2008.
- [11] T. Dyba et al., "What works for whom, where, when, and why? On the role of context in empirical software engineering," in *Proc. the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2012, pp. 19-28.
- [12] K. Philippe, "Voyage in the agile memplex," *Queue*, vol. 5, no. 5, pp. 1, 2007.
- [13] T. Zimmermann and N. Nagappan, "Predicting defects with program dependencies," in *Proc. 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, IEEE, pp. 435-438.
- [14] M. E. Conway, "How do committees invent?" *Datamation*, vol. 14, no. 4, pp. 28-31, 1968.
- [15] H. D. James and R. E. Grinter, "Splitting the organization and integrating the code: Conway's law revisited," in *Proc. the 21st International Conference on Software Engineering*, 1999, ACM, pp. 85-95.
- [16] P. Ralph, P. Johnson, and H. Jordan, "Report on the first SEMAT workshop on general theory of software engineering (GTSE 2012)," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 2 pp. 26-28, 2013.
- [17] J. Pontus, P. Ralph, M. Goedicke, P. W. Ng, K. J. Stol, K. Smolander, I. Exman, and D. E. Perry, "Report on the Second SEMAT Workshop on General Theory of Software Engineering (GTSE 2013)," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 47-50, 2013.
- [18] SEMAT (Software Engineering Method and Theory). [Online]. Available: <http://www.semat.org>
- [19] I. Jacobson, S. H. Huang, M. K. Mattsson, P. McMahon, and E. Seymour, "Semat—Three year vision," *Programming and Computer Software*, vol. 38, no. 1, pp 1-12, Springer January 2012.
- [20] Essence-OMG Submission. [Online]. Available: <http://www.omg.org/cgi-bin/doc?ad/2012-11-01>
- [21] I. Jacobson, P. W. Ng, P. McMahon, I. Spence, and S. Lidman, "The essence of software engineering: the SEMAT kernel," *Communications of the ACM*, no. 10, pp. 42-49, 2012.
- [22] I. Jacobson, P.-W. Ng, McMahon, and S. Lidman, *The Essence of Software Engineering: Applying the SEMAT kernel*, Addison-Wesley, 2013.
- [23] P. W. Ng and S. H. Huang, "Essence: A Framework to Help Bridge the Gap between Software Engineering Education and Industry Needs."
- [24] T. Sedano and C. Péraire, "State-based Monitoring and Goal-driven Project Steering: Field Study of the SEMAT Essence Framework," 2013.
- [25] L. Osterweil, "Software processes are software too," in *Proc. the 9th International Conference on Software Engineering*, IEEE Computer Society Press, 1987, pp. 2-13.
- [26] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting Software Architectures: Views and Beyond*, Pearson Education, 2002.
- [27] B. P. Kruchten, "The 4+1 view model of architecture," *Software, IEEE* 12, no. 6, pp. 42-50, 1995.
- [28] I. Jacobson and P.W. Ng, *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*, Addison-Wesley Professional, 2004.
- [29] Intooitus. [Online]. Available: <http://www.intooitus.com/products/infusion>
- [30] Unitask. [Online]. Available: <http://www.unitask.com/oracledaily/2012/08/26/jsf-component-libraries-quality-is-more-than-zero-bugs/>
- [31] E. W. Dijkstra, *A Discipline of Programming*, vol. 1. Englewood Cliffs: prentice-hall, 1976.
- [32] E. W. Dijkstra, "On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective*, Springer New York, 1982, pp. 60-66.
- [33] L. Kurt, *Field Theory in Social Science: Selected Theoretical Papers*, Dorwin Cartwright (ed), London: Tavistock, 1952.

Pan-Wei Ng received his Ph.D. from Nanyang Technological University, Singapore. He is currently working for Ivar Jacobson International. He is a software engineering coach and advisor to large organizations in Asia Pacific helping them with scaled agile transformations, product line engineering, lean and agile, requirements, architecture and testing. He is the author of "Aspect Oriented Software Development with Use Cases" (2004) and "The Essence of Software Engineering: Applying the SEMAT Kernel" (2013).