Situational Method Quality

Liming Zhu and Mark Staples
1 NICTA, Australian Technology Park, Eveleigh, NSW 1430, Australia
2 School of Computer Science and Engineering
University of New South Wales, NSW, Australia
[Liming.Zhu, Mark.Staples]@nicta.com.au
WWW home page: http://www.cse.unsw.edu.au/~limingz/

Abstract. Some overall method characteristics, such as agility and scalability, have become increasingly important. These characteristics are different from existing method requirements which focus on the functional purposes of individual method chunks and overall methods. Characteristics like agility and scalability are often not embodied in the function of a single method chunk but are instead reflected in constraints over one or more method chunks, connections between method chunks and cross-cutting aspects of the overall method. We propose the concept of method tactics, which are techniques for achieving certain method quality attributes. We identify a list of method tactics focusing on agility and scalability by considering factors that affect these quality attributes. We validate the feasibility of using method tactics by applying them to traditional software development method chunks and deriving practices for agile development. We examine the effectiveness of the tactics by comparing our derived practices with existing practices for agile development. The comparison results show that most of the derived practices are found in existing agile methods. We also identify new practices that may have potential for use in agile methods. The results demonstrate initial support for our proposal for the use of method tactics, and for the extraction or invention of further cross-cutting primitive method tactics for more flexible situational method engineering.

1 Introduction

Method quality is often considered to be functional conformance to method requirements or industry best practices. However, in system development, quality is defined not only in terms of correctness (conformance to functional requirements) but also satisfaction of non-functional requirements. Non-functional requirements are equally important for method engineering. Some examples of non-functional characteristics for methods include:

Please use the following format when citing this chapter:

Zhu, L., Staples, M., 2007, in IFIP International Federation for Information Processing, Volume 244, Situational Method Engineering: Fundamentals and Experiences, eds. Ralyté, J., Brinkkemper, S., Henderson-Sellers B., (Boston Springer), pp. 193-206.

- Agility: the ability of a method to accommodate expected change rapidly and efficiently
- Scalability: the ability of a method to retain its effectiveness with larger (or smaller) team size and product size
- Interoperability: the ability a method to interact with other methods and environments
- Usability: the ease of use of the method by human agents to achieve the goals supported by the method

The current approach to improve quality characteristics of a method is to tailor an existing method or select and integrate existing method chunks [20] that posses some degree of the desired quality characteristic. For example, in the software development domain, existing agile practices can be added to a method, or a particular agile method can be tailored.

However, this approach has some limitations. A method engineer should be able to directly and systematically improve specific quality attributes of a method, rather then rely solely on selecting and integrating method chunks from among existing practices. This can not be achieved if method engineers do not understand the underlying reasons why an existing method chunk supports those quality attributes. This limits the flexibility and precision of a method engineer's ability to improve the quality characteristics of a method.

In this paper, we propose a new concept called "method tactic". A method tactic is a technique for method engineering, intended to achieve specific method qualities. Method tactics can apply to an existing method chunk, a collection of method chunks, or an entire method. Although some method tactics can themselves be realized as method chunks, usually method tactics manifest as constraints over a method chunk, or more frequently as cross-cutting constraints over multiple method chunks. We observe that the cross-cutting nature of these tactics also makes it difficult to treat them as a single method chunk in a method repository. Thus this approach complements existing approaches for method engineering that rely mostly on selecting method chunks from a method repository.

We have collected an initial collection of tactics for agility and scalability. We have not intended to collect a complete set of such tactics or to rigorously categorize them. Our goal has been to identify some practical techniques that a method engineer can use to improve the non-functional quality of a method. We have conducted an initial validation of our method tactics by applying them to software development methods. Using the method tactics we have been able to derive practices that exhibit the desired method qualities and that match industry best practices and methods that promote the same qualities. We have also been able to identify new practices that have not yet been included in software development processes. These new practices can be further empirically validated and considered as candidates to be included in future software development methods. Our work has a number of contributions:

- Method tactics characterize why methods achieve specific method quality characteristics.
- Applying method tactics directly allows more flexible method design, and can potentially identify new reusable method chunks as best practices.

- A method tactic may affect multiple method quality characteristics in different directions. Our approach makes such trade-offs explicitly understood for resolution by method engineers. For example, the trade-offs between agility and scalability in software development methods is an area of growing interest [5, 12].

This paper is organized as follows. We first discuss related work in section 2. In section 3, we introduce the concept of method tactics and illustrate them by providing a list of tactics that affect agility and scalability. In section 4, we apply these tactics to general activities (method chunks) in software development processes in order to achieve specific method characteristics. We demonstrate that the derived practices closely resemble existing software development practices that promote those characteristics. We discuss the limitations of our work in section 5, and present conclusions and future work in section 6.

2 Related Work

Methods possess both functional and non-functional characteristics. Situational method requirements often specify non-functional requirements of methods, such as being able to handle large team size, large project size, high product requirement volatility, fast responsiveness to change, and flexibility.

Many approaches to method engineering focus on assembly techniques [6, 8, 13, 21, 24]. Such approaches propose strategies such as association and integration to bridge or merge method chunks [20], and use configuration packages [11] during assembly and adapting. Such approaches do not include atomic means to achieve cross-cutting concerns such as method qualities. Our work addresses this issue by identifying and using method tactics, which complement method chunks and bridging/merging-based assembly techniques.

Method engineering theories have been successfully applied to software development domains to create situation and project specific methods [1-3, 9, 17]. Assembly approaches have been used to investigate support for product qualities, such as system interoperability [19]. Certain method quality attributes, such as agility has been investigated [10, 22] and compared among methods. However, the agility of these methods has only been analyzed at the phase and practice level. The atomic and primitive reasons why these practices are "agile" has not been explicitly captured and analyzed. Our work is the first attempt to extract these underlying reasons and to use them in the context of method engineering.

The concept of tactics for design is not new. Atomic architectural and design tactics have been used to achieve non-functional cross-cutting product quality at the architectural level [4]. These tactics have been useful because most non-functional product requirements can not be achieved by selecting and assembling functional components. The analogy between products and methods (processes) is well-recognized [18]: a method should be designed to satisfy method requirements [20] just as products are designed to satisfy product requirements. By further following this analogy, we observe that applying the concept of tactics to method engineering can provide atomic means of achieving cross-cutting method quality.

3 Method Tactics

As defined previously, a method tactic is a technique for method engineering, intended to achieve specific method qualities. Ideally, it should be possible to systematically analyze a specific quality of a method by using a method quality reasoning model. Such a model would represent how a method tactic could manipulate parameters leading to the quality, and would help to explain the effectiveness of such tactics. However, no formal method for reasoning about method quality exists. Nonetheless, some informal factors can be identified.

In this paper, we use the method qualities of agility and scalability as illustrative examples. We have chosen from the software development methodology literature a number of well-recognized factors that affect these two qualities:

- Efficiency of information flow (speed, responsiveness and leanness)
- Type of feedback
- Frequency of activity/feedback/auditing
- Incremental completion of tasks
- Reversible actions
- Task interdependency

By inventing techniques to try to affect these factors, we have identified a preliminary list of method tactics. This list is not intended to be a complete list. As expected, most tactics affect multiple method qualities in different directions. That is, by achieving one quality method engineers may have to sacrifice another quality. The analysis of method tactics that we present using these informal factors should be taken as general analyses – we note where counter-examples may exist in some circumstances. In presenting our list of method tactics, we have grouped them for ease of analysis.

Method Tactic: Use verbal communication and "light" informal documentation
Method Tactic: Use formal documentation

Verbal communication can increase the agility of a method through increased speed of information flow, responsiveness and leanness. Relying primarily on verbal communication does not necessarily remove documentation completely. Many industry practices that promote verbal communication tactics are also conveyed as practices about using less documentation. "Light" informal documentation can include forms such as email or instant messaging logs, and wiki pages. Verbal communication can suffer from poor scalability to larger team size and longer project durations. Purely verbal communication on complex topics among a large number of people is not highly effective, and informal documentation is prone to obsolescence through poor maintenance. Longer project durations present an increased risk of higher personnel turnover, leading to a decrease in the effectiveness of organizational memory and knowledge. Verbal communication and informal documentation can also negatively affect method reliability. Verbal communication and informal documentation is often used in smaller projects and for methods that require extremely high agility in terms of responsiveness and leanness. Formal documented communication manifests the opposite quality attributes. Extensive and rigorous

documentation usually decreases method agility but can increase reliability and scalability. (However, counter-examples to this can exist if documentation is poorly maintained.)

Method Tactic: Downstream-driven input (feed-back)
Method Tactic: Upstream-driven input (feed-forward)

The inputs into an activity can be based on downstream activities/artifacts or upstream activities/artifacts. For example, in the software development method context, design activities could rely solely on the upstream requirements to verify design output against the requirements. However, design activities could also rely on inputs from downstream activities such as coding (Code Smell) or testing (Design for Test).

Downstream-driven inputs effectively establish a feedback loop. Downstream feedback can provide rich information that is quite different to that from upstream activities. Exiting an activity with low quality outputs may lead to higher downstream or overall cost due to unnecessary rework not caused by changing environment and requirements. In such situations, the longer and less frequent the feedback loop, the greater the overall cost. Thus, downstream-driven input is often used with a very short loop and with frequent feedback.

This tactic may not scale well for large systems when rework cost is not linear due to the effects of complexity. The cost of rework may outweigh the richer feedback obtained through "trialing" downstream activities. On the other hand, the value of downstream feedback may decrease when a domain is very mature or a team is very experienced. This is why large projects in mature domains often still follow waterfall methods to some degree, with less frequent iterations.

Method Tactic: Introduce continuous feedback/auditing
Method Tactic: Introduce staged feedback/auditing

Continuous feedback is effectively a very small feedback loop between interconnected method chunks. For example, in software development methods, it is possible to maintain such feedback loops between designing and coding or between coding and testing. Continuous feedback improves agility tremendously. In terms of scalability, it works well if the activities within a loop are performed by one individual. However, if it involves multiple people, the communication overhead, synchronization issues and potential resource contention will harm the scalability of the method. For example, continuous integration in software development involves a coding/building continuous feedback loop. This can suffer from scalability due to the reasons mentioned above.

Method Tactic: Allow a single method chunk to be carried out incrementally

Method chunks may be carried out incrementally to fit with tactics for iteration or feedback. However, there are other reasons for incremental execution. For example, requirements change volatility can lead to a risk that early work might be rendered obsolete. Incremental execution allows certain decisions to be deferred until required. Just-in-time elaboration and maximizing work-not-to-be-done are practices that employ this tactic. Some product properties have an emergent nature (especially

in large scale systems [16]) and are difficult to plan. Incremental execution can be useful in these situations.

This tactic improves method agility but may suffer from scalability over the long run on certain activities. Long-term incremental refinement of large products may reach a breaking point that can only be solved by a comprehensive overhaul [5].

Method Tactic: Use configuration management

In order to allow reversible changes, configuration management should be used to track all changes. This tactic increases method agility by enabling managed changes. Configuration management may introduce additional cost and overhead. However, for any project involving multiple working in parallel on the same artifacts, the benefits of configuration management normally outweigh its costs.

Tactics such as configuration management clearly have it's a cross-cutting aspect. Introducing a single method chunk called "configuration management" won't work. Although initial method chunks may be required to plan and establish a configuration management environment, configuration controls influence many existing method chunks, for example to support "check-out" and "check-in" activities before and after their execution.

Method Tactic: Reduce task dependencies between multiple resources.

Task dependency between multiple resources introduces communication and synchronization overhead. Method chunks should be designed to support maximum parallelism not only among method chunks but also among instances of a single method chunk. Although the nature of the task often has a major impact on its ability to be partitioned and executed in parallel, process analysis techniques can improve this. As we have demonstrated, method tactics can be applied in different ways:

- One can add constraints to a single method chunk. For example, to make an activity shorter or allow it to be carried out incrementally.
- One can add constraints to a block of method chunks. For example, to introduce continuous feedback loop or fixed iteration times in an area of method chunks.
- One can add method elements to many method chunks to realize cross-cutting tactics. For example, to apply configuration management activities to each of the method chunks in a method that are affected by configuration control policies.

Figure 1 shows relationships between method tactics and method chunks in method engineering. Tactics can also be applied to instances of method or method chunks. None of these method transformations can be easily modeled using traditional method chunks and integration/association-based assembly techniques.

4 Applying Method Tactics to Software Development Methods

In order to validate the feasibility and effectiveness of using method tactics, we now apply method tactics to general software development method chunks. We have only selected tactics listed in section 3 that improve method agility. Some of these also

harm scalability. The generic method chunks include eliciting and defining requirements (R), design (D), coding (C), testing (T), and product and project management (P). This is shown in Table 1. Each cell represents the practice derived from applying the corresponding method tactic to the generic activity. Multiple related method chunks can be involved during the process due to the cross-cutting nature of the tactics and non-functional method requirements. We then determine if the derived practices match existing agile practices in industry, and consider if they suffer from poor scalability. We looked at eight agile methods:

1. Feature Driven Development (FDD)
2. Extreme Programming (XP)
3. Dynamic Systems Development Method (DSDM)
4. Scrum (Scrum)
5. Agile Software Development (ASD)
6. Crystal (Crystal)
7. Lean Software (Lean)
8. Agile RUP (ARUP)

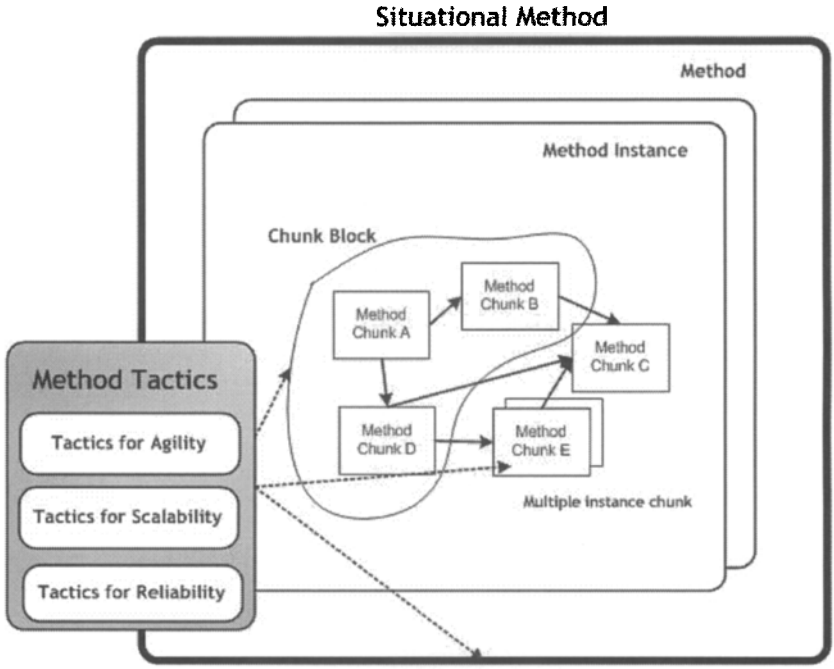

Figure 1 Method Tactics

Due to a high degree of overlap, we only include the first four in the following report. All eight methods were considered when we try to identify potential missing practices.

Table 1. Applying method tactics to generic software development method chunks

	R	D	C	T	P
1 Verbal communication	1xR	1xD	1xC	1xT	1xP
2 Downstream driven feedback	2xR	2xD	2xC	2xT	2xP
3 Frequent feedback/checking	3xR	3xD	3xC	3xT	3xP
4 Incremental completion	4xR	4xD	4xC	4xT	4xP
5 configuration management					
6 Reduce task dependency					

1xR: Use verbal communication and informally documented requirements
Existing Agile Practices: On-Site Customer (XP), User Stories (XP), Active User Involvement (DSDM), Collaboration and Cooperation among stakeholders (DSDM), Domain Object Modeling (FDD)

Applying verbal communication to requirements means relying less on formally documented requirements. Customers tell stories and the stories are recorded and not immediately scrutinized. Such methods rely on the on-site-customer to elaborate requirements verbally in a just-in-time fashion.

1xD: Use verbal communication and informally documented design
Existing agile practices: Pair Programming (XP), Code as Design Documentation (XP), Metaphor (XP)

1xC: Use verbal communication or lightly documented code
Existing agile practices: Pair Programming (XP), Code as Documentation (XP)

Most agile methods promote lightly documented design and use standard-complying self-documenting code as the main design artefact. Since design is essentially integrated with coding, pair programming also acts as a way of communicating designs.

1xT: Use verbal communication and informally documented tests
Existing agile practices: Unit Testing by Developer (XP), Pair Programming (XP)

The goal of verbal communication is to reduce communication overhead between resources. Making a single resource perform multiple functional goals reduces communication overhead to zero since only one person is involved. Making developers do unit testing is an example of applying this tactic.

1xP: Use verbal communication and informally documented products
Existing agile practices: Co-location (XP), On-site Customer (XP), Code as Documentation (XP)

When this tactic is applied cross-cuttingly to the overall project, co-location is the result. However, providing co-location and on-site customers is difficult for large projects, and in global development and outsourcing contexts.

2xR: Design/Coding/Testing/Product driven requirements validation with user
Existing agile practices: Just-in-time requirements elaboration (XP – design/coding driven), Testable Requirements (XP –testing driven), Short Release (XP –product driven), Productionizing (XP – product driven), Frequent Product Delivery (DSDM – product driven)

The downstream activities for requirements are design, coding, testing and product. We consider them separately as indicated in the brackets. Products are considered have the richest feedback because: 1) productionizing leads to more intermediate steps to be carried out which may reveal more issues; and 2) products can be used to seek feedback from users directly. The second point is especially important for requirements activities. However, the cost of constantly producing working and tested products can be costly for large projects. The trade-off between cost and agility should be considered here.

2xD: Coding/Testing/Product driven design validation
Existing agile practices: Spiking (XP – coding driven), Code Smell (XP – coding driven), Design for Test (XP – testing driven),
Potential missing practices: Release driven design review, non-functional requirements validation through design review

2xC: Testing/Product driven coding
Existing agile practices: Test Driven Development (XP – testing driven), Continuous/Nightly Build (XP – product driven)

2xT: Product driven testing
Existing agile practices: Acceptance Testing (XP)

Coding and designing are often intertwined activities in agile processes. Although testing is extensively used for driving all upstream activities (requirements, design and coding), there is no emphasis on using the product feedback to improve design and coding directly. Products are often used as a way to elicit feature-based feedback rather than systematic non-functional requirements, e.g. performance, reliability, scalability validation through design review. We identify this as a potential missing agile practice.

3xR: Continues requirement validation
Existing agile practices: On-Site Customer (XP), Active User Involvement (DSDM), Collaboration and Cooperation among stakeholders (DSDM), Domain Object Modeling (FDD)

3xD: Continuous design validation
Existing agile practices: Refactoring/Code Smell (XP)

3xC: Continuous code validation
Existing agile practices: Test Driven Development (XP), Pair Programming (XP), Regression Testing (XP), Continuous Integration (XP)

The continuous method tactic can be applied in two ways:

1. By creating immediate feedback loop between adjacent method chunks. Refactoring and test driven development are two examples.
2. By adding extra resources on the same task to provide immediate feedback. Pair programming and on-site customers are two examples.

Continuous feedback will improve software development agility. The cost and scalability of this very short feedback loop is affected by whether multiple people are involved. For example, one benefit of continuous integration is that messages about build failures can be sent to the specific individual(s) who caused the failure. The entire development team need not be involved. However, continuous integration may not scale well due to resource contention on build servers and synchronization issues during a long build. Adding extra resources is also costly and can only be justified in certain circumstances. For example, the increased cost of pair programming is often justified by mentoring and training benefits.

3xT: Continuous test code validation
Existing agile practices: Pair Programming (XP)
Potential missing practices: Continuous test quality check, test refactoring, test smell.

It has been argued that test driven development will produce code as good as the test design. Since test code is not as rigorously examined as other part of the development, low quality code could be produced due to low quality test code. Currently, test design largely depends on experience. Employing more rigorous test design techniques or auto-test-generation can mitigate the risks involved.

3xP: Continues product validation
Existing agile practices: Short Release (XP), Productionizing (XP), Sprint/Sprint Review (Scrum), Frequent Product Delivery (DSDM)

Continuous product validation has appeared in almost all agile methods. However, it is not exactly continuous, but instead usually a very short release iteration. Different methods put different constraints or use different criteria for the length of iterations.

4xR: Incremental requirement definition
Existing agile practices: Just-in-time requirement elaboration (XP), Accept Requirement Changes (all agile methods)

4xD: Incremental design
Existing agile practices: YAGNI/Simple Design (XP), Refactoring (XP)

4xC: Incremental coding

Existing agile practices: Short Release (XP), Developing by Feature (FDD), Iterative and Incremental development (DSDM), Product Backlog (Scrum)

4xT: Incremental testing
Existing agile practices: N/A
Potential missing practices: Simple Testing.

Incremental execution of method chunks has been an important practice in all agile methods due to a number of reasons:

- Constantly changing environment and requirements
- Emergent properties instead of planned properties of a system [16]
- Inevitable programming rework [7]

The focus of incremental work has been on all activities except testing. This might be due to the fact that testing is essential in quality assurance. However, it has been observed that one difficulty in test driven development is the amount of time taken to setup testing infrastructure, including high quality skeletons, stubs and mock objects. Due to the high volatility of requirements change and design refactoring (which affects interfaces), testing code can become obsolete quickly. There is a need to balance the sophistication of the testing code and its ability to perform high quality testing. The XP YAGNI principle (You Aren't Gonna Need It) can be applied cautiously to progressively improve the coverage and quality of testing code.

4xP: Incremental product development
Existing agile practices: Short Release (XP), Developing by Feature (FDD), Iterative and Incremental development (DSDM), Product Backlog (Scrum)

Tactics such as configuration management and task dependency are overall cross-cutting tactics. Thus, we try to apply them to the overall method.

5: Configuration management
Existing agile practices: Reversible Changes (DSDM), Configuration Management (FDD).

6: Task Dependencies:
Existing agile practices: Collective ownership (XP)
Potential missing practices: Architecture driven process planning

Task dependency in software development is very much related to the architecture of a product [27]. Architecture is usually not systematically used to optimize method parallelism and concurrent development. However, there has been some preliminary research [15] into the issue.

As discussed in section 3, some tactics promoting agility suffer from scalability issues. The software development agile practices we derived here inherit these scalability issues. Some issues can be mitigated [12]. Others have to be accepted on balance [5].

Overall there is a high degree of fit between our derived practices and existing industry practices. Variants or different elaborations exist in industry for most of our

derived practices, but they achieve agility through the same means. This supports our claim that method tactics can be and should be discovered and used in situational method design. By applying these method tactics systematically to relevant method chunks, we can potentially identify new practices. The quality trade-offs documented for each tactic can be used to analyze practice trade-offs within new situations. Certain situations may exacerbate quality problems while others may make them less relevant. We have demonstrated that agility and scalability trade-offs in software development methods can be better understood through method tactic analysis.

5 Discussion

The discovery and accumulation of method tactics should be based on both theoretically sound grounds and also empirical observation and validation. Because there is little existing theory on non-functional method quality, we have conducted our initial work by observing important factors in practices within existing software development methods. Similar general observations have also been made in other specific development domains such as product line development [23] and COTS-based development [14]. However, the observations are usually too high-level to be useful in validating fine-grained practice-level method chunks. There are a number of limitations of our work due to this.

- Our list of tactics may appear to be arbitrary in terms of their orthogonality and level of abstraction. Some of the tactics are overlapping and some others have close relationships. Some tactics may be able to be divided into more atomic ones. This limitation could be addressed by the development and validation of reasoning models and parameters for each method quality attribute. Then, method tactics could be organized around their influence on these parameters. We are currently working on establishing such reasoning models.
- Our list may omit some important kinds of tactics, especially those used in other method domains. We are looking into other method engineering domains and may expand our use of method tactics to these broader domains.

6 Conclusion

Just like a product, a method has to be designed to satisfy situational requirements. These situational requirements include both functional requirements and non-functional requirements. Achieving scalability, agility, reliability and usability of a method is equally important as achieving functional requirements. We observe such non functional requirements can often be achieved only through using cross-cutting techniques rather than changing or adding single method chunks. We propose the concept of method tactics to capture these cross-cutting techniques. Our preliminary work identified a number of such tactics for achieving agility and scalability. Most of the tactics affect both -ilities in different directions. This raises interesting trade-off analysis opportunities in situational method design. We validated these general tactics by applying them to general software development methods. The result

demonstrates that agile practices can be designed intentionally and these derived agile practices match existing agile methodologies. This opens a new door to designing new method chunks in more flexible and creative ways. We plan to visualize these tactics and affected development processes in process definition languages such as Little-JIL[25] or goal-oriented languages such as i*[26]. We also plan to include a more systematic evaluation framework to evaluate newly proposed techniques which claim certain cross-cutting ilities.

Acknowledgements

NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

References

1. M. N. Aydin and F. Harmsen, "Making a Method Work for a Project Situation in the Context of CMM," in *Product-Focused Software Process Improvement (PROFES)*, 2002 pp. 158-171.
2. M. Bajec, R. Rupnik, and M. Krisper, "A Framework for Reengineering Software Development Methods " in *International Conference on Software Engineering Advances (ICSEA'06)*, 2006 p. 28.
3. M. Bajec, D. Vavpotič, and M. Krisper, "Practice-driven approach for creating project-specific software development methods," *Information and Software Technology,* vol. 49(4), pp. 345-365, 2007.
4. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2 ed.: Addison-Wesley, 2003.
5. B. W. Boehm and R. Turner, *Balancing agility and discipline : a guide for the perplexed.* Boston: Addison-Wesley, 2003.
6. S. Brinkkemper, M. Saeki, and F. Harmsen, "Assembly Techniques for Method Engineering," in *10th International Conference Advanced Information Systems Engineering (CAiSE'98)*, 1998.
7. A. Cass and L. Osterweil, "Programming Rework in Software Processes," Department of Computer Science, University of Massachusetts UM-CS-2002-025, 2002.
8. E. Domínguez and M. A. Zapata, "Noesis: Towards a situational method engineering technique," *Information Systems,* vol. 32(2), pp. 181-222, 2007.
9. B. Henderson-Sellers and C. Gonzalez-Perez, "A comparison of four process metamodels and the creation of a new generic standard," *Information and Software Technology,* vol. 47, pp. 49-65, 2005.
10. B. Henderson-Sellers and A. Qumer, "An Evaluation of the Degree of Agility in Six Agile Methods and its Applicability for Method Engineering," *Information and Software Technology,* vol. In Press, 2007.
11. F. Karlsson and P. Agerfalk, "Method configuration: adapting to situational characteristics while creating reusable assets " *Information and Software Technology,* vol. 46(9), pp. 619-633, 2004.

12. D. Leffingwell, *Scaling software agility : best practices for large enterprises*. Upper Saddle River, NJ: Addison-Wesley, 2007.
13. I. Mirbel and J. Ralyte, "Situational Method Engineering: Combining Assembly-based and Roadmap-driven Approaches," *Requirements Engineering,* vol. 11(1), pp. 58-78, 2006.
14. F. Navarrete, P. Botella, and X. Franch, "Reconciling Agility and Discipline in COTS Selection Processes " in *the Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07)*, 2007.
15. M. Nonaka, L. Zhu, M. A. Barbar, and M. Staples, "Project Delay Variability Simulation in Software Product Line Development,," in *International Conference on Software Process (ICSP'07) co-located with ICSE'07*, 2007.
16. L. Northrop, R. Kazman, M. Klein, D. Schmidt, K. Wallnau, and K. Sullivan, "Ultra-Large Scale Systems: The Software Challenge of the Future," 2006.
17. B. Nuseibeh, A. Finkelstein, and J. Kramer, "Method engineering for multi-perspective software development," *Information and Software Technology,* vol. 38(4), pp. 267-274, 1998.
18. L. Osterweil, "Software Processes Are Software Too," in *International Conference on Software Engineering (ICSE)*, 1987.
19. J. Ralyte, P. Backlund, H. Kuhn, and M. Jeusfeld, "Method Chunks for Interoperability," in *International Conference on Conceptual Modeling (ER)*, 2006.
20. J. Ralyte, R. Deneckere, and C. Rolland, "Towards a Generic Model for Situational Method Engineering," in *International Conference Advanced Information Systems Engineering (CAiSE'03)*, 2003.
21. M. Rossi, J.-P. Tolvanen, B. Ramesh, K. Lyytinen, and J. Kaipala, "Method Rationale in Method Engineering," in *33rd Hawaii International Conference on System Sciences (HICSS)*, 2000.
22. M. K. Serour and B. Henderson-Sellers, "Introducing Agility: A Case Study of Situational Method Engineering Using the OPEN Process Framework," in *28th Annual International Computer Software and Applications Conference (COMPSAC '04)*, 2004.
23. K. Tian and K. Cooper, "Agile and Software Product Line Methods: Are They So Different?," in *the First International Workshop on Agile Product Line Engineering (APLE'06)*, 2006.
24. I. v. d. Weerd, S. Brinkkemper, J. Souer, and J. Versendaal, "A Situational Implementation Method for Web-based Content Management System-applications: Method Engineering and Validation in Practice," *Software Process Improvement and Practice,* vol. 11, pp. 521-538, 2006.
25. A. Wise, "Little-JIL 1.5 Language Report," Department of Computer Science, University of Massachusetts, Amherst, MA 2006.
26. E. Yu, "Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering," in *the Third International Symposium on Requirements Engineering (RE'97)*, 1997.
27. L. Zhu, R. Jeffery, M. Huo, and T. T. Tran, "Effects of Architecture and Technical Development Process on Micro-Process," in *International Conference on Software Process (ICSP'07) co-located with ICSE'07*, 2007.