Chapter XVII
Agile Method Fragments and Construction Validation

Q.N.N. Tran
University of Technology, Sydney, Australia

B. Henderson-Sellers
University of Technology, Sydney, Australia

I. Hawryszkiewycz
University of Technology, Sydney, Australia

ABSTRACT

Method fragments for work units and workflows are identified for the support of agile methodologies. Using one such situational method engineering approach, the OPEN Process Framework, we show how the full set of these newly identified agile method fragments, each created from the relevant powertype pattern as standardized in the Australian Standard methodology metamodel of AS 4651, can be used to recreate four of the currently available agile methods: XP, Scrum, and two members of the Crystal family—thus providing an initial validation of the approach and the specifically proposed method fragments for agile software development.

INTRODUCTION

Situational method engineering (Welke & Kumar, 1991) is the subdiscipline of software engineering in which methodologies (a.k.a. methods) are envisaged as being constructed from parts called method fragments. These are identified from best practice and stored in a repository (or method base). In this chapter, we identify new fragments to support the work units and work flows needed to support agile software development—a set of method fragments that completes and complements those described in Tran, Henderson-Sellers, and Hawryszkiewycz (2007). These fragments are compliant with the metamodel of the Australian Standard AS4651 and extend the existing repository of the OPEN Process Framework (OPF: Firesmith & Henderson-Sellers, 2002; http://www.opfro.org), chosen on the basis of it having the most extensive content in its method base. Following this discussion of new fragments, we then use them to recreate a number of existing agile methods as validation of both the SME ap-

proach and the particular set of newly proposed agile method fragments.

AGILE SOFTWARE DEVELOPMENT AND THE AS4651 METAMODEL

As discussed in Tran et al. (2007), agile development adheres to the following fundamental values (Agile Manifesto, 2001):

- **Individuals and interactions** should be more important than processes and tools.
- **Working software** should be more important than comprehensive documentation.
- **Customer collaboration** should be more important than contract negotiation.
- **Responding to change** should be more important than following a plan and has a strong focus on teamwork.

Here, we use the basic understanding of agile software development to identify method fragments compatible with the OPEN Process Framework and the metamodel described in AS4651 (Standards Australia, 2004). The overall architecture of this metamodel is shown in Figure 1, using the notion of a powertype (Odell, 1994) (for full details see Tran et al., 2007).

We will now describe in more detail the metaclasses that are relevant to agile fragments but which were not covered in Tran et al. (2007), that is, work units (tasks and techniques) and workflows (a third kind of work unit)—the focus of this chapter.

WorkUnit-Related Metaclasses

A WorkUnit is a job performed within a project. A WorkUnitKind is a specific kind of work unit, characterized by its purpose within the project. It is specialized into TaskKind, TechniqueKind, and WorkFlowKind.

A task is a small-grained work unit that focuses on what must be done in order to achieve a given purpose. A TaskKind is a specific kind of task, characterized by its purpose within the project.

A technique is a small-grained work unit that focuses on how the given purpose may be achieved. A TechniqueKind is a specific kind of technique, characterized by its purpose within the project.

WorkFlow-Related Metaclasses

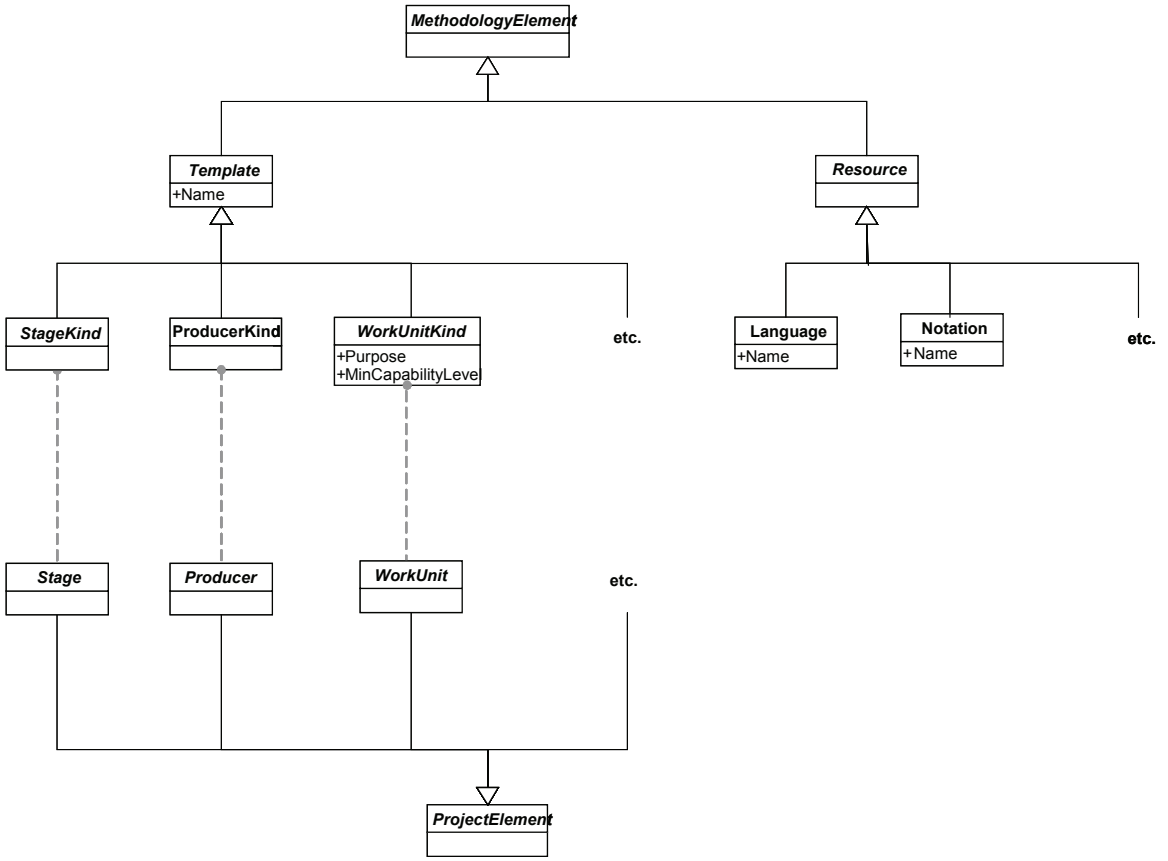
A WorkFlow is a large-grained work unit that operates within a given area of expertise. A WorkFlowKind is a specific kind of work flow, characterized by the area of expertise in which it occurs. It is specialized into ActivityKind and ProcessKind. An activity is a work flow that represents a continuous responsibility. An ActivityKind is a specific kind of activity, characterized by the area of expertise in which it occurs. A process is a work flow that represents a discrete job. A ProcessKind is a specific kind of process, characterized by the area of expertise in which it occurs.

Here, we first evaluate what current support is available for a range of agile methods for these two groups of metaclasses and their generated fragments. When the support is not available (in terms of a fragment held in the methodbase), we propose the addition of a new fragment, documented in the OPF standard style including alphabetical ordering (see Appendices).

NEWLY IDENTIFIED FRAGMENTS TO SUPPORT AGILE DEVELOPMENT

This study has identified a large number of new work unit fragments that could be considered for addition to the current OPF repository/methodbase. These are summarized in the following sections and are detailed in Appendixes A-C in terms of the metaclass from which they are generated.

Figure 1. Overall architecture of AS 4651



Work Unit Fragments

There are three major kinds of work unit kinds in the metamodel from which fragments have been generated. These are discussed in turn: TaskKind fragments, TechniqueKind fragments, and two sorts of WorkFlowKind fragments.

TaskKind

Existing TaskKind fragments include analyze technology, describe application, elicit requirements, prototype the architecture, develop iteration plan, code, refactor, integrate software, write manuals, and prepare other documentation, document the design as well as several risk assessment and management tasks, testing tasks, and teamwork building and management tasks.

However, there are new tasks associated with agile methods that are not encompassed by these agile-focussed tasks in the existing repository. These are documented in Appendix A for addition to the repository.

TechniqueKind

As one might anticipate, the support for agile *techniques* in the existing OPF repository is incomplete, although some minimal descriptions of pair programming, planning game, system metaphor, and refactoring were made in Henderson-Sellers (2001). In addition to those already documented (Firesmith & Henderson-Sellers, 2002)—regression testing, acceptance testing, beta testing, unit testing, team building, role assignment, group problem solving, brainstorming and workshops—the recommended

new fragments to describe agile-specific techniques are found in Appendix B (those mentioned in Henderson-Sellers, 2001 are also included there in full, expanded detail for the sake of completeness).

WorkFlowKind

ActivityKind

In addition to OPF's build, evolutionary development, user review, consolidation, and project management activity kinds, we propose one additional fragment: team management activity kind (as detailed in Appendix C).

ProcessKind

The phases described in the current OPF repository appear to be sufficient to support all the agile methods studied to date.

The "build activity kind" for XP development (see ActivityKind section) can alternately be replaced by the aggregation of:

- "Requirement engineering process kind;"
- "Designing process kind;"
- "Implementation process kind;"
- "Testing process kind;"
- "Deployment process kind;" and
- "Evaluation process kind"

given that the process of each kind is repeated in short iterative cycles.

Note: Team building may be viewed as a process. However, the work involved in team building has been covered by "human resource management" sub-activity kind of "project management" activity kind of OPEN (particularly, staffing and task allocation).

RECREATING AGILE METHODS

Process construction can be undertaken in one of several ways. Three of these (maps, activity diagrams, and deontic matrices) are compared in Seidita, Ralyté, Henderson-Sellers, Cossentino, and Arni-Bloch (2007). Of these, the one recommended for use with the OPF is that of deontic matrices, first introduced in MOSES (Henderson-Sellers & Edwards, 1994) and SOMA (Graham, 1995). A deontic matrix is a two dimensional matrix of values that represent the possible or likely relationship between each pair of method fragments in the OPF. The actual values depend upon a number of factors such as project size, organizational culture, domain of the application to be developed, and the skills and preferences of the development team. Once completed, they give guidance on the most appropriate selection of method fragments. Although five levels are suggested in the literature, for a specific project often binary values are used. In addition, increasingly sophisticated teams will wish to see their process mature commensurately. Using the deontic matrix approach, new method fragments are easily added to an organization's existing methodology. In this approach, there is a deontic matrix to link activities to tasks, tasks to techniques, producers to tasks, and so forth.

In the following subsections we outline the results of the use of these deontic matrices and list the fragments selected for inclusion in each of the four respective methodologies: XP (Table 1), Scrum (Table 2), Crystal Clear (Table 3), and Crystal Orange (Table 4).

DISCUSSION OF THE RECREATED AGILE METHODS

For each of the four agile methods, we have identified all the fragments from the enhanced OPF methodbase necessary for their recreation. Each approach is seen to be different (as one might expect) supporting the application of a method engineering approach to construct *situational* methodologies. It is thus reasonable to propose that the fragments created and documented in this chapter are complete and adequate to support (at least these four) agile methodologies.

Table 1. Assembly of method fragments to reproduce XP Methodology

ACTORS			
Teams	Peer programming team XP-style team		
Roles	Agile programmer Agile customer XP tester Tracker Coach Consultant Project manager		
PROCESS			
Lifecycle	XP lifecycle		
Builds	Release Iteration		
Milestones	Code drop Iteration completed First release completed Subsequent release completed		
Documents	Story cards Release plan Iteration plan Test set of documents		
Software Items	Software components (source code, running code, test software)		
Tools			
Phases, tasks and techniques	*Initiation phase*	*Tasks:* Write user stories Explore architectural possibilities Analyze technologies Describe application Prototype the architecture Develop release plan Develop iteration plan	*Techniques:* Agile team building Small/short release System metaphor Planning game Iteration planning game Stand-up meeting
	Construction phase	*Activity*: Build *Tasks*: Design agile code Code Refactor Testing tasks Integrate software	*Techniques:* Pair programming Simple design Refactoring Continuous integration Collective ownership Open workspace Regression testing Acceptance testing Unit testing Stand-up meeting
	Delivery phase	*Activity*: Build (in shorter cycles; more focused on "user review" and "consolidation" sub-activities) *Tasks*: Design agile code Code Refactor Testing tasks	*Techniques:* Same as above (more usage of testing techniques, e.g., acceptance testing)
	Usage phase	Write manuals and prepare other documentation	

Table 2. Assembly of method fragments to reproduce Scrum Methodology

ACTORS			
Teams	Scrum team		
Roles	Scrum master Product owner Common feveloper roles Agile customer		
PROCESS			
Lifecycle	Scrum lifecycle		
Builds	Release Sprint		
Milestones	Sprint completed Release completed		
Documents	Product backlog Release plan Iteration plan (i.e., sprint backlog) Test set of documents		
Software Items	Software components (source code, running code, test software)		
Tools			
Phases, tasks and techniques	*Pregame phase*	*Tasks:* Elicit requirements Develop release plan Risk assessment and management Team building	*Techniques:* Agile team building
	Development/game phase	*Activity*: Build *Tasks*: Design agile code Code Testing tasks Integrate software	*Techniques:* Sprint planning meeting Daily Scrum meeting Sprint/iteration review meeting Testing techniques
	Postgame phase	*Tasks*: Integrate system Write manuals and prepare other documentation	*Techniques:*

Table 3. Assembly of method fragments to reproduce Crystal Clear Methodology

ACTORS		
Teams	Project team	
Roles	Designer Programmer Agile customer Sponsor Requirements engineer Project manager	
PROCESS		
Lifecycle	Iterative, incremental, parallel	
Builds	Release Iteration	
Milestones	Iteration completed Release completed Various technical milestones within each iteration, for example, start of requirements engineering, first design review, first user review, final user review, pass into test…	
Documents	System requirements specification (use case specification or feature descriptions) User's manual Templates Standards (e.g., for coding, regression testing) Design sketches Test set of documents	
Software Items	Software components (running code, migration code, test software)	
Tools	upperCASE tools (configuration management and documentation management tools) lowerCASE tools (compiler) Groupware tools (e.g., printing whiteboard)	
Phases, tasks and techniques	(For each iteration): *Activity*: Build *Tasks*: Design tasks Code Testing tasks Document the design Monitor work product	*Techniques* Small/short releases (2-3 months) Testing techniques Monitoring by progress and stability Parallelism and flux Reflection workshop Methodology tuning

Table 4. Assembly of method fragments to reproduce Crystal Orange Methodology

ACTORS		
Teams	Function team (equivalent to the composition of OPEN's software requirements team, software architecture team, and software development team) Infrastructure team (equivalent to the composition of OPEN's system development team and hardware development team) Test team System planning team (equivalent to OPEN's project initiation team or management team) Project monitoring team (equivalent to OPEN's management team) Architecture team Technology teams (one per each speciality, equivalent to OPEN's user interface team, peer programming team, database team, documentation team, or test team)	
Roles	Project manager/big boss Requirements engineer Architect Designer Programmer Tester Technical writer Tester User Business expert (equivalent to a type of stakeholder such as "manager") Technical facilitator, design mentor (equivalent to tracker, coach, or Scrum master)	
PROCESS		
Lifecycle	Iterative, incremental, parallel	
Builds	Release Iteration	
Milestones	Iteration completed Release completed Various technical milestones within each iteration, for example, start of requirements engineering, first design review, first user review, final user review, pass into test…	
Documents	System requirements specification Design set of documents (e.g., UI design, object model, database design etc.) Schedules (equivalent to release plan and iteration plan documents) Status report User's manual Templates Standards (e.g., for coding, regression testing, notation, design, and quality) Test set of documents	
Software Items	Software components (source code, running code, migration code, test software)	
Tool	upperCASE tools (configuration management, documentation management, and modeling tools) lowerCASE tools (compiler) Groupware tools (e.g., printing whiteboard, team progress tracking, team communication)	
Phases, tasks and techniques	Develop release plan Develop iteration plan (For each iteration): *Activity*: Build *Tasks*: Design tasks Code Testing tasks Document the design Monitor work product	*Techniques* Small/short release (3-4 months) Individual ownership Testing techniques Sprint/iteration review Monitoring by progress and stability Parallelism and flux Holistic diversity strategy Reflection workshop Methodology tuning

SUMMARY, CONCLUSION, AND FURTHER WORK

Using the concepts embodied in the situational method engineering approach, we have argued that existing method bases are deficient in their support for agile methodologies. Complementing and extending the set of method fragments outlined by Tran et al. (2007) in the preceding chapter, we have described fragments relevant to work units and workflows. Together, these additions comprise a set of method fragments each of which is created from the relevant powertype pattern as standardized in the Australian Standard methodology metamodel of AS 4651 (Standards Australia, 2004). Finally, we have validated this approach by recreating four agile methods: XP (Table 1), Scrum (Table 2), and two members of the Crystal family (Tables 3 and 4).

ACKNOWLEDGMENT

We wish to thank Dr. Cesar Gonzalez-Perez for his useful comments on an earlier draft of this chapter. We also wish to thank the Australian Research Council for funding under Discovery Grant DP0345114.

REFERENCES

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). *Agile software development methods, reviews and analysis* (VTT Publication no 478). Espoo, Finland: VTT.

Adams, S. G. (2003). Building successful student teams in the engineering classroom. *Journal of STEM Education Innovations and Research, 4*(3&4).

AgileManifesto. (2001). *Manifesto for agile software development*. Retrieved March 14, 2005, from http://www.agilemanifesto.org/

Auer, K., & Miller, R. (2001). *XP applied*. Boston: Addison Wesley.

Barker, J., Tjosvold, D. et al. (1988). Conflict approaches of effective and ineffective project managers: a field study in a matrix organization. *Journal of Management Studies, 25*(2), 167-177.

Beck, K. (1999). Embracing change with extreme programming. *IEEE Computer, 32*(10), 70-77.

Beck, K. (2000). *Extreme programming explained: Embrace change.* Boston: Addison-Wesley.

Beck, K. (2003). *Test driven development—by example*. Boston: Addison Wesley.

Bens, I. (1997). Facilitating conflicts. In M. Goldman (Ed.), *Facilitating with ease!* (pp. 83-108). Sarasota, FL, USA: Participative Dynamics.

Bloom, P. J. (2000). *Circle of influence: Implementing shared decision making and participative management.* Lake Forest, USA: New Horizons.

Capozzoli, T. K. (1995). Conflict resolution: a key ingredient in successful teams. *Supervision, 56*(12), 3-5.

Cockburn, A. (1998). *Surviving object-oriented projects—a manager's guide.* Boston: Addison-Wesley.

Cockburn, A. (2002a). *Agile software development.* Boston: Addison-Wesley.

Cockburn, A. (2002b). Agile software development joins the "would-be crowd." *Cutter IT Journal, 15*(1), 6-12.

Cockburn, A., & Highsmith, J. (2001). Agile software development: the people factor. *IEEE Computer, 34*(11), 131-133.

Cohen, C. F., Birkin, S. J., et al. (2004). Managing conflict in software testing. *Communications of the ACM, 47*(1), 76-81.

Coram, M., & Bohner, S. (2005). The impact of agile methods on software project management.

In *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05).*

Corn, M., & Ford, D. (2003). Introducing an agile process to an organization. *IEEE Computer, 36*(6), 74-78.

Couger, D., & Smith, D.C. (1992a). Evaluating the motivating environment in South Africa compared to the United States—Part 1. *South African Computer Journal, 6*.

Couger, D., & Smith, D. C. (1992b). Evaluating the motivating environment in South Africa compared to the United States—Part 2. *South African Computer Journal, 8*.

Firesmith, D. G., & Henderson-Sellers, B. (2002). *The OPEN Process Framework. An introduction*. London: Addison-Wesley.

Fisher, K., Rayner, S., & Belgard, W. (1995). *Tips for teams: a ready reference for solving common team problems.* New York: MacGraw-Hill, Inc.

Fowler, M. (2001). Is design dead? In G. Succhi & M. Marchesi (Eds.), *Extreme programming examined* (pp. 3-7). Boston: Addison-Wesley.

Graham, I. M. (1995). *Migrating to object technology.* Wokingham, UK: Addison-Wesley.

Grazier, P. (1998). *Team motivation: Ideas to energize any team*. TeambuildingInc.com.

Henderson-Sellers, B. (2001). Enhancing the OPF repository. *JOOP/ROAD, 14*(4), 10-12, 22.

Henderson-Sellers, B., & Edwards, J. M. (1994). *BOOKTWO of object-oriented knowledge: The working object.* Sydney, NSW, Australia: Prentice-Hall.

Highsmith, J. (2000). Extreme programming. *CUTTER Consortium's Agile Project Management Advisory Service.*

Jarboe, S. (1996). Procedures for enhancing group decision making. In B. Hirokawa and M. Poole (Eds.), *Communication and group decision making* (pp. 345-383). Thousand Oaks, CA, USA: Sage Publications.

Jeffries, R. (2004). *Where's the spec, the big picture, the design?* Retrieved on August 15, 2005, from http://www.xprogramming.com/xpmag/docBigPictureAndSpec.htm

Kearny, L. (1995). *The facilitator's toolkit: tools and techniques for generating ideas and making decisions in groups.* Amherst, USA: Human Resource Development Press.

Kezsbom, D. S. (1992). Bringing order to chaos: Pinpointing sources of conflict in the nineties. *Cost Engineering, 34*(11), 9-16.

Kim, M. O. (2002). Coping with conflict in concurrent design environment. *ACM SIGGROUP Bulletin, 23*(1), 20-23.

McDaniel, G., Littlejohn, S., et al. (1998). A team conflict mediation process that really works! In *Proceedings of the 9th International Conference on Work Teams, Dallas, USA.*

Millis, B. J., & Cottell, P. G. (1997). *Cooperative learning for higher education*. Phoenix, AZ, USA: Oryx Press.

Odell, J. J. (1994). Power types. *Journal of Object-Oriented Programming, 7*(2), 8-12.

Paulsen, D. (2004). Leadership essentials: facilitation skills for improving group effectiveness. In *Proceedings of the 32nd Annual ACM SIGUCCS Conference on User services*, Baltimore, USA.

Qumer, A., & Henderson-Sellers, B. (in press). An evaluation of the degree of agility in six agile methods and its applicability for method engineering. *Information and Software Technology.*

Rayeski, E., & Bryant, J. D. (1994). Team resolution process: a guideline for teams to manage conflict, performance and discipline. In M. Beyerlein & M. Bullock (Eds.), *Proceedings of the International Conference on Work Teams Proceedings: Anni-*

versary Collection. The Best of 1990-1994 (pp. 215-221). Denton, TX, USA: University of North Texas, Center for the Study of Work Teams.

Rees, F. (1998). *The facilitator excellence handbook: Helping people work creatively and productively together.* San Francisco: Jossey-Bass/Pfeiffer.

Rees, F. (2001). *How to lead work teams: Facilitation skills*. San Francisco: Jossey-Bass/Pfeiffer.

Seidita, V., Ralyté, J., Henderson-Sellers, B., Cossentino, M., & Arni-Bloch, N. (in press). A comparison of deontic matrices, maps and activity diagrams for the construction of situational methods. In J. Eder, S.L. Tomassen, A.L. Opdahl, & G. Sindre (Eds.), (pp. 85-88).

Serour, M. K., Henderson-Sellers, B., & Dagher, L. (2006). Augmenting an existing software development process with a team building activity: a case study. In Z. Irani, O.D. Sarikas, J. Llopis, R. Gonzalez, & J. Gasco (Eds.), *Proceedings of the European and Mediterranean Conference on Information Systems 2006 (EMCIS2006)*, CD. West London, UK: Brunel University.

Sibbet, D. (2002). *Principles of facilitation: the purpose and potential of leading group.* San Francisco: The Grove Consultants International.

Soller, A. L. (2001). Supporting social interaction in an intelligent collaborative learning system. *International Journal of Artificial Intelligence in Education 12*(1), 40-62.

Standards Australia. (2004). *Standard metamodel for software development methodologies—AS 4651-2004,* Sydney, NSW, Australia: Standards Australia.

Tran, Q. N. N., Henderson-Sellers, B., & Hawryszkiewycz, I. (2007). Some method fragments for agile software development. In M. Syed (Ed.), *Handbook of research on modern systems analysis and design technologies.* Hershey, PA, USA: IGI.

Trimmer, K. J., Collins, R. W., Will, R. P., & Blanton, J. E. (2000). Information systems development: can there be "good" conflict? In *Proceedings of the 2000 ACM SIGCPR Conference on Computer personnel research,* Chicago, USA.

Wake, W. C. (2001). *Extreme programming explored*. Boston: Addison Wesley.

Webne-Behrman, H. (2005). *Conflict resolution*. Retrieved July 31, 2005, from http://www.ohrd.wisc.edu/onlinetraining/resolution/index.asp

Welke, R., & Kumar, K. (1991). Method engineering: a proposal for situation-specific methodology construction. In W. W. Cotterman & J. A. Senn (Eds.), *Systems analysis and design: A research agenda*. Chichester, UK: Wiley.

Wikipedia. (2005). *Extreme programming.* Retrieved August 10, 2005, from http://en.wikipedia.org/wiki/Extreme_Programming

KEY TERMS

Agile Method: A method that is people focused, flexible, speedy, lean, responsive, and supports learning (based on Qumer & Henderson-Sellers, 2007).

Agility: *Agility is a persistent behaviour or ability of a sensitive entity that exhibits flexibility to accommodate expected or unexpected changes rapidly, follows the shortest time span, uses economical, simple and quality instruments in a dynamic environment, and applies updated prior knowledge and experience to learn from the internal and external environment.* (Qumer & Henderson-Sellers, 2007)

Metamodel: A model of models.

Method Engineering: The engineering discipline to design, construct, and adapt methods, techniques, and tools for systems development.

Method Fragment: Construction of a software development method for a specific situation.

Producer: An agent that executes work units.

ProducerKind: a specific kind of producer, characterized by its area of expertise.

Stage: A managed time frame within a project.

StageKind: A specific kind of stage, characterized by the abstraction level at which it works on the project and the result that it aims to produce.

Task: A small-grained work unit that focuses on what must be done in order to achieve a given purpose.

TaskKind: A specific kind of task, characterized by its purpose within the project.

Technique: A small-grained work unit that focuses on how the given purpose may be achieved.

TechniqueKind: A specific kind of technique, characterized by its purpose within the project

WorkProduct: An artefact of interest for the project.

WorkProductKind: A specific kind of work product, characterized by the nature of its contents and the intention behind its usage.

WorkUnit: A job performed within a project.

WorkUnitKind: A specific kind of work unit, characterized by its purpose within the project.

ENDNOTES

[1] See "Simple design" technique and TechniqueKind section.

[2] These are existing OPEN tasks fragments listed in the TaskKind section.

[3] See TechniqueKind section.

[4] See OPEN's technique "regression test"

APPENDIX A. TASK KINDS

Design Agile Code

- *Purpose*: Designing is creating a structure that organizes the logic of the system, which is eventually implemented by code (Beck, 2000). The "design agile code" task refers to the designing task in an *agile* process.

It may appear that agile processes, for example, XP, involve only coding without designing (Jeffries, 2004). Actually, an agile process involves a lot of design, although it does it in a very different way from the conventional development processes (Beck, 2000; Fowler, 2001). This is because the cycle of requirements specification, designing and implementation is much more rapid in an agile project than in a conventional plan-driven process (Jeffries, 2004). Thus, the task of design in an agile process (which we name "design agile code" task) adopts the following unique practices:

1. Do not separately document designs in formal graphical models or specifications as in the conventional processes. Instead, capture the design primarily in the code itself. Thus, design is part of the coding process: design generated are described by code and implemented by code. The team communicates the design through the code. Pictures and documents are only generated when it is deemed really necessary (Fowler, 2001; Jeffries, 2004).
2. Follow an "evolutionary, incremental, and continuous" design approach, instead of a "planned design" approach as in an conventional process (Highsmith, 2000; Fowler, 2001). The evolutionary design approach means that the design of the system grows as the system is implemented. The design will change as the program evolves. This indicates

again that design needs to be part of the coding process. On the other hand, in planned design, agilists argue that the designers make all major design decisions in advance, generating a complete system design before handing it off to programmers to code. Agile developers are against this planned design approach because it involves high cost in dealing deal with changing requirements.

When performing the "design agile code" task, the "simple design" technique should be used (see TechniqueKind section).

The "design agile code" task must be performed in conjunction with the "refactor" task (this is an existing OPEN's TaskKind). Each time a new requirement emerges and a new piece of code needs to be designed and added to the existing program, the previous design of (the affected part of) the existing program should be revisited and improved (if necessary) to ensure that the new system (once the new code is added) will exhibit a "simple design"[1]—that is, the existing code should be refactored before newly designed code is added (Beck, 2000). The coupling between "refactor" task and "design agile code" task allow the evolutionary design mechanism to be carried out in a systematic, easy-to-manage and effective manner, rather than in an ad-hoc manner of design restructuring.

The "design agile code" task should also be coupled with testing tasks,[2] testing techniques, and "continuous integration" technique,[3] so as to ensure that the newly designed and added code still keeps the system in synchronization (Highsmith, 2000).

Develop Release Plan

- *Purpose*: Plan for a forthcoming release, including determining *what requirements/ features* are included in the release and *when* it is going to be delivered.

This task generates a "release plan document" (see DocumentKind section in Tran et al., 2007). Both developers and customers and both parties need to agree on the developed plan. Release planning can be conducted using the planning game technique (see TechniqueKind section). Release planning initiates, and can be affected by, iteration planning.

Explore Architectural Possibilities

- *Purpose*: Explore the possible architectural configurations for the target system. This task is performed in the "exploration phase" of the XP development process.

The programmers in an XP team can perform this task by spending a week or two building a system similar to what they will build eventually, but doing it in three or four ways. Different programmer pairs can try the system different ways and compare them to see which one feels best, or two pairs can try the system the same way and see what differences emerge. This task may involve building a prototype for each possible architectural configuration; cross reference: task "prototype the architecture" of OPEN.

Architectural explorations are most important when the customer comes up with stories that the XP team has no idea how to implement.

Monitor Work Products

- *Purpose*: Monitor the development progress of each work product of each team. This task should be performed throughout an iteration/release of an agile development process. "monitoring by progress and stability" can be used.

Team Management Task Kinds

a. "Manage shared artefacts" task kind: consisting of the following sub-tasks:

- "**Identify shared artefacts"**: is the task of identifying artefacts that can be accessed mutually by different teams or different members in a team.
- "**Allocate shared artefacts":** is the task of allocating these shared artefacts to teams, roles, and/or tasks.
- "**Specify Permissions to Shared Artefacts":** is the task of defining the permissions granted to each different role/team to access the artefact.

b. "Mediate/monitor the performance of team's tasks" task kind: consisting of the following subtasks:

- "**Meditate/monitor team's interactions":** is the task of enforcing/monitoring interactions between team members during the task. Each team member's interaction protocols should depend on his/her roles.
- "**Conflict management"**: is the task of identifying and resolving conflicts between team members.
- "**Monitor members' performance"**: is the task of assessing how well a team member is performing the task and (probably) providing private/public feedback to reinforce team members' focus on task
- "**Member motivation"**: is the task of encouraging team members' participation in tasks (e.g., by initiating & facilitating round-robin participation or role switching around the team)
- "**Ensure workload balance":** is the task of detecting workload imbalance amongst team members and taking actions to balance the workload.

c. "Specify team policies" task kind:

- *Purpose*: Identify and document the policies (or rules or conventions) that govern the collaborative work within a particular team.

d. "Specify team structure" task kind:

- *Purpose*: Define the structure of a particular team in terms of:

- Roles that make up the team (both SE roles and teamwork roles)
- Acquaintance relationships amongst these roles
- Authority relationships that govern these acquaintances

Note that in agile development, the team structure can frequently change.

Write User Stories

Purpose: Allow customers (users of the system) to specify their requirements. This task should be considered as a new sub-task of the existing OPEN's task "elicit requirements." It is to be performed by customers in an XP project team (see "story card document kind" in the DocumentKind section in Tran et al., 2007).

Ideally, the feature/requirements described by each story should be able to be accomplished within 1-5 programming weeks. Stories should be testable.

Other team members (e.g., programmers) should give copious and quick feedback to the first few stories written by the customer, so that the customer can learn quickly how much ground to cover in each story and what information to include and exclude. A story should be specified in such a way that the programmers can confidently estimate the effort needed to provide the feature required by the story.

APPENDIX B. TECHNIQUE KINDS

Agile Team Building

- *Purpose*: Assist in the building of project teams for agile projects, for example, XP-style team

Table 1. Comparison of team support in four agile methods

	XP	Scrum	Crystal Clear	Crystal Orange
Number of teams	1 team per project	1-4 or more	1	1-10
Team size	3-16	5-9	1-6	1-6

(see TeamKind section in Tran et al., 2007).

- *Description*: Since agile development relies substantially on teamwork, collaboration and communication, the team is the key for success (Coram & Bohner, 2005). Consequently, special care must be given to the building of project teams (Serour, Henderson-Sellers& Dagher, 2006).

Agile project teams have various important characteristics that distinguish them from traditional OO-development teams. As such, the conventional practices for team building (e.g., OPEN's "team building" technique) are not sufficient (or inappropriate) to the building of agile teams.

The following suggestions should be considered when forming project teams, specifically for agile development:

- *Small number and size* (Coram & Bohner, 2005): There should be a small number of teams per project and a small number of members per team. This ranges from a single team of 3-16 developers on XP to up to six teams of 2-6 members on DSDM (see Table 1). Small teams are required to foster collaboration and are more likely to require less process and planning to coordinate team members' activities.
- *High competence of members*: Agile development derives much of its agility by relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans (Cockburn, 2002a, b). In addition, the productivity difference between the best and worst programmers on a team would surface most clearly when the members are working on tasks essential to software delivery, which make up most of an agile process (note that agile processes strip non-essential activities from projects, leaving developers more tasks on software delivery) (Corn & Ford, 2003). Thus, Boehm's principle of top talent, "use better and fewer people" (Cockburn & Highsmith, 2001), is central to an agile process (Corn & Ford, 2003), and high competency of team members is a critical factor in agile projects' success (Cockburn & Highsmith, 2001). In all agile methodologies (e.g., XP, Scrum, ASD, and DSDM), the emphasis on team members' talent, skill, and knowledge is evident. Too many slow workers either slow the entire team or end up left behind by their faster teammates.
- *Good rapport amongst team members* (Coram & Bohner, 2005): Developers who do not work well together, or a single strong-willed developer, could each destroy the collaborative nature of the team. Low team rapport represents a significant risk for an agile project. A successful agile team is one which is highly cohesive and mutually supportive.
- *Low risks from high turnover* (Coram &Bohner, 2005): High turnover of team members can lead to loss of critical knowledge. The project manager should consider this risk when examining whether a team is right for an agile project. To retain relevant knowledge, appropriately skilled and knowledgeable members should be retained when building (or changing) teams.
- *Co-location of members* (Corn & Ford, 2003): Teams using agile processes tend to make decisions more quickly than plan-driven

teams, relying on more frequent (and usually informal) communication to support this pace. Thus, team members should try to avoid distributed development for at least the first 2 or 3 months after initiating an agile process. If distributed developers must be combined, the team leader/project manager should bring as many people as possible together for the first week or two of the project can increase the likelihood of success. Crystal, Scrum, and ASD all advocate close and direct collaboration practices including barrier-free collocated teams.

- Minimal interaction and dependencies between different teams and maximal cohesion within each team.

Collective Ownership

- *Purpose*: Facilitate sharing of responsibilities of teams' outcomes, support sharing of knowledge amongst team members and promote the code's quality.
- *Description*: In XP, everybody takes responsibility for the whole of the system. Any team member who sees an opportunity to add value to any portion of the code is required to do so at any time. This technique is opposed to other two models of code ownership: no ownership and individual ownership. In the former model, nobody owns any particular piece of code. If someone wants to change some code, he/she can do it to suit his own purpose, whether it fits well with what is already there or now. The result is chaos. The code grows quickly but it also quickly grows unstable. With individual ownership, the only person who can change a piece of code is its official owner. Anyone else who sees that the code needs changing has to submit his request to the owner. The result is that the code diverges from the team's understanding, as people are reluctant to interrupt the code owner.

Conflict Resolution

- *Purpose*: Manage conflicts between team members in teamwork.
- *Description*: There are numerous techniques for conflict resolution in the literature. Only some are discussed here.

Webne-Behrman (2005) proposes an 8-step resolution process that can be employed by a team member to effectively managing conflict in teamwork. These steps will not guarantee an agreement, but they greatly improve the likelihood that the problems can be understood, solutions explored, and consideration of the advantages of a negotiated agreement can occur within a relatively constructive environment.

1. "Know thyself" and take care of self
2. Clarify personal needs threatened by the dispute
3. Identify a safe place for negotiation
4. Take a listening stance into the interaction
5. Assert your needs clearly and specifically
6. Approach problem-solving with flexibility
7. Manage impasse with calmness, patience, and respect
8. Build an agreement that works

Rayeski and Bryant (1994) suggest the use of the team resolution process for managing conflict in teams. The process allows the team to address conflict as it occurs, thus providing the team with self-sufficient methods for handling disagreement on their own. Rayeski and Bryant's process includes three steps:

1. Collaboration: Initially as conflict arises, it should be handled informally between the two team members in a private setting.
2. Mediation: If the situation escalates, a mediator is brought into the dispute to assist both sides in reaching an agreement (e.g., the team leader or task facilitator).

3. Team counseling: If efforts of collaboration and mediation fail, team counseling is held at a team meeting, with all members of the team present.

Other resources for team conflict resolution:

- Generic teams (Barker, Tjosvold et al., 1988; Kezsbom, 1992; Fisher, Rayner, & Belgard, 1995; Capozzoli, 1995; Bens, 1997; McDaniel, Littlejohn et al., 1998; Rees, 1998).
- IS teams (Trimmer, Collins, Will, & Blanton, 2000; Kim, 2002; Cohen, Birkin et al., 2004).

Continuous Integration

- *Purpose*: Promote correctness of systems/programs and support early discovery of errors/defects.
- *Description*: With the "continuous integration" technique, developers integrate their new codes into a baseline system/program and run a set of regression tests[4] on it until it is 100% correct (Beck, 2000; Coram & Bohner, 2005). This should be done after every few hours or a day of development at most.

Continuous integration helps to promote quality of the developed system/program because errors caused by a change can be quickly discovered and it is also obvious who should fix the errors—the developers who produce the change. Finding defects early also reduces the effort of fixing them.

However, the downside of this technique is that developers must write a comprehensive set of tests to be used as regression tests and must take the time to integrate and test their code. This may require a shift in developer perspective if the developer is accustomed to simply writing code that is then tested by a different group (Coram & Bohner, 2005).

Daily Meeting

- *Purpose*: Encourage frequent communication amongst team members about their work progress; Promote team-based, rapid, intense, cooperative, and courteous development; identify and remove impediments to development process.
- *Description*: These meetings are held daily in a short period of time (15-30 min.) when all team members meet with each other in a room and each takes turn to tell the group:
  - What he or she accomplished the prior day
  - What he or she plans to do today
  - Any obstacles or difficulties he or she is experiencing

The time and location of the daily meeting should be constant. Any team members working from remote locations can join via conference phones.

Someone should be responsible for keeping the daily meetings short (by enforcing the rules and making sure people speak briefly) and promoting the productivity of the meeting as much as possible. In XP, this person can be "coach," while in Scrum, the "Scrum master" is usually selected.

While others (such as managers and customers) may attend the daily meetings, only those directly involved the work such as developers, coach, Scrum master, and product owner can speak.

Team members should arrange themselves in a circle, generally around a focus such as a table. People not on the team should sit/stand outside the team's circle.

In XP projects, the daily meetings are referred to as "stand-up meetings," since everyone is required to stand (standing is intentional to motivate the team to keep the meeting short) (Auer & Miller, 2001). Often the pair-programming pairs are dynamically formed during the daily meeting as the tasks for the day are discussed. Two programmers that are best equipped to handle the task join together.

In Scrum projects, the daily meetings are called "daily Scrum meeting." An important activity of the Scrum master during daily Scrum meetings is to record and remove any impediment that is

reported by a team member, for example, slow network/server, uncertain technology use, uncertain design decision, over-loaded individual responsibilities, and so forth. If a Scrum master cannot make a decision on how to remove an impediment during the meeting, he/she is responsible for making a decision and communicating it to the whole team within one hour after the meeting.

If there are multiple teams, a daily "Scrum of Scrums" can be organized, where "Scrum masters" from each Scrum team meet after the daily Scrums for their own daily Scrum.

Holistic Diversity Strategy (Cockburn, 1998)

- *Purpose*: Assist in the building of project teams for agile projects.
- *Description*: Holistic diversity strategy suggests that for each function (or set of functions) to be delivered, a small team consisting of 2-5 members from *mixed, different specialties* should be formed, to be responsible for delivering that function. For example, a team can include different specialists, each from requirements gathering, UI design, technical design, programming, or testing. The team should be evaluated as a unit, so there is no benefit to hiding within a specialty. The team members should be co-located so they can communicate directly, instead of by writing. There should be no documentation within the team, although the team will have documentation responsibilities to the rest of the project.

Iteration Planning Game

- *Purpose*: Assist planning for an iteration in XP (see "develop iteration plan" in TaskKind section).
- *Description*: The iteration planning game is similar to the planning game in that cards are used as the pieces. This time, though, the pieces are task cards instead of story cards. The players are all he individual programmers. The timeframe is iteration (1 to 4 weeks) instead of release. The phases and moves are similar.
- *Exploration phase*:
  - Write a task: Programmers take the stories for the iteration and turn them into tasks.
  - Split a task/combine tasks: If a task cannot be estimated at a few days, it should be broken down into smaller tasks. If several tasks each take an hour, they should be combined to form a larger task.
- *Commitment phase*:
  - Accept a task: A programmer accepts responsibility for a task.
  - Estimate a task: The responsible programmer estimates the number of ideal engineering days to implement each task.
  - Set load factors: Each programmer chooses their load factor for the iteration—the percentage of time they will spend actually developing.
  - Balancing: Each programmer adds up their task estimates and multiplies by their load factor. Programmers who turn out to be overcommitted must give up some tasks.
- *Steering phase*:
  - Implement a task: A programmer takes a task card, finds a partner (i.e., pair programmer), writes the test cases for the task, makes them all work, then integrates and releases the new code when the universal test suite runs.
  - Record progress: Every 2 or 3 days, one member of the team asks each programmer how long they have spent on each of their tasks and how many days they have left.

- Recovery: A programmer who turns out to be overcommitted, asks for help by reducing the scope of some tasks, asking the customer to reduce the scope of some stories, shedding non-essential tasks, getting more or better help, and asking the customer to defer some stories to a later iteration
- Verify story: When functional tests are ready and the tasks for a story are complete, the functional tests are run to verify that the story works.

Methodology-Tuning Technique (Cockburn, 2002a)

- *Purpose*: Facilitate on-the-fly methodology construction and tuning.
- *Description*: The technique supports methodology construction and tuning by suggesting what to do at five different times in a project: right away, at the start of the project, in the middle of the first release, after each release, and in the middle of subsequent releases.

- Right away (regardless when it is in any project): Discover the strengths and weaknesses of the development organization through short interviews. People to be interviewed may be the project manager, team leaders, designers, and/or programmers. Questions to ask include a short history of the project, work products, what should be changed next time, what should be repeated next time, and project's priorities.
- At the start of the project: Have two or more people working together on creating/selecting a base methodology for the project, for example, XP, RUP, or Crystal. Then, hold a team meeting to discuss the base methodology's workflow and conventions, and tailor it to the corporate methodological standards, producing a starter methodology.
- In the middle of the first release: Run a small interview with team members (individually or in a group), asking whether the team is going to be successful in the way that they are working. The goal is not to change the whole starter methodology if the team is not working (unless it is catastrophically broken). The aim is to get safely to the first release delivery.
- After each release: Hold a reflection workshop (see technique "reflection workshop"). The two questions to be asked are what the team learned and what can they do better. Very often the team would tighten standards, streamline the workflow, increase testing, and reorganize the team structure.
- In the middle of the subsequent increments: Hold interviews or a reflection workshop to think of new and improved ways of working.

Monitoring by Progress and Stability (Cockburn, 2002a)

- *Purpose*: Monitor each work product of each team throughout an iteration/release.
- *Description*: Each team's work product should be monitored with respect to *both* progress and stability (Cockburn, 1998). Progress is measured in milestones, which are sequential; the stability states are not necessarily sequential.

Progress milestones:

1. Start
2. Review 1
3. Review 2
4. To test
5. Deliver

Stability states:

1. Wildly fluctuating
2. Fluctuating

3. Stable enough to review

A common sequence of stability states might be 1-2-3-2-3-2-3. A deliverable rates "to test" may get re-labeled as "fluctuating" if some unexpected problem were encountered that questioned the design or the requirements.

Open Workspace

- *Purpose*: Facilitate physical proximity in teamwork, particularly for agile teams.
- *Description*: The project team works in a large room with small cubicles around the periphery. Pair programmers work on computers set up in the centre (Beck, 1999). With this work environment setting, it is much easier for any team member to get help if needed, just by calling across the room. In the long term the team benefits from the intense communication. The open workspace helps pair programming to work, and the communication aids all the practices.

Pair Programming

- *Purpose*: Promote communication between team members, productivity of members, and quality of resulting products in an agile development. This technique is used in XP methodology.
- *Description*: Pair programming is a programming technique where two people program with one computer, one keyboard, and one monitor (Beck, 2000). It should be noted that pair programming is not about one person programming while another person watches. Pair programming is a dialogue between two people trying to simultaneously program (and analyze and design and test) and understand together how to program better. It is a conversation at many levels, assisted by and focused on a computer.

In XP, pairing is dynamic. If two people pair in the morning, they might be paired with other developers at different times during the day.

Sometimes pairs contain one partner with much more experience than the other partner. If this is true, the first few sections will look a lot like tutoring. The junior partner will ask lost of questions and type very little. However, in a couple of months, typically, the gap between the partners is not nearly as noticeable as it was at first.

Pair programming is particularly suitable to agile development because it encourages communication. When an important new bit of information is learned by someone on the team, this information can be quickly disseminated throughout the team, since the pairs switch around all the time. The information actually becomes richer and more intense as it spreads and is enriched by the experience and insight of everyone on the team. Pair programming is also often more productive and results in higher quality code than if the work is divided between two developers and then the results are integrated.

Parallelism and Flux (Cockburn, 2002a)

- *Purpose*: Maximize the parallelism in the production of an iteration/release, while permitting changes in the work products.
- *Description*: This technique can easily be followed if the above technique, "monitoring by progress and stability" has been followed. According to the "parallelism and flux" technique, any dependent task can start as soon as all the predecessor work products are in the "stable enough to review" stability state. While the predecessor work products are still "wildly fluctuating," the performers of the successor tasks can start working out their basic needs, but should not start serious design/coding. For example, as soon as the system requirements are "stable enough to review," designs can begin parallel design. Similarly, as soon as the design reaches "stable enough

to review" status, serious programming can begin (although tentative programming may have been done earlier, to assist in creating the design).

Planning Game

- *Purpose*: Assist planning for a release in XP (see "develop release plan" in TaskKind section).
- *Description*: The planning game offers a set of rules that remind everyone in an XP project of how they should act during planning for a release. The rules serve as an aid for the building of a trusting and mutually respectful relationship between the customer and the XP project team. The rules of the planning game are as follows.
- *The goal*: the goal of the game is to maximize the value of software produced by the team. From the value of the software, the cost of its development and the risk incurred during development need to be determined.
- *The pieces*: The pieces in the planning game are the story cards (see "story card" in DocumentKind section in Tran et al., 2007).
- *The players*: Two players in the planning game are development and business. Development consists collectively of all the people who will be responsible for implementing the system, for example, programmers, coaches, trackers, and project managers. Business consists collectively of all the people who will make the decisions about what the system is supposed to do, for example, customers, real users of the project, and sales people.
- *The moves*: There are three phases to the game.
    - *Exploration phase*: The purpose of this phase is to give both development and business players an appreciation for what the system should eventually do. Exploration has three moves:
        - Write a story: Business players write a story describing something the system needs to do (see "story card" in DocumentKind section in Tran et al., 2007).
        - Estimate a story: Development players estimate how long the story will take to implement.
        - Split a story: If development players cannot estimate a whole story, or if business players realize that part of a story is more important than the rest, business players can split a story into two or more stories.
    - *Commitment phase*: The purpose of this phase is for business players to choose the scope and date of the next release, and for development players to confidently commit to delivering it. The commitment phase has four moves.
        - Sort by value: Business players sort the stories into three piles: 1) those without which the system will not function, 2) those that are less essential but provide significant business value, and 3) those that would be nice to have.
        - Sort by risk: Development players sort the stories into three piles: 1) those that they can estimate precisely, 2) those that they can estimate reasonably well, and 3) those that they cannot estimate at all.
        - Set velocity: Development players tell business how fast the team can program in ideal engineering time per calendar month.
        - Choose scope: Business players choose the set of story cards in the release.
    - *Steering phase*: The purpose of this phase is to update the plan based on what is learned by development and business

players. The steering phase has four moves.

- Iteration: At the beginning of each iteration, business players pick one iteration worth of the most valuable stories to be implemented.
- Recovery: If development players realize that they has overestimated their velocity, they can ask business players what is the most valuable set of stories to retain in the current release based on the new velocity and estimates.
- New story: If business players realize they need a new story during the middle of the release, they can write the story. Development players estimate the story, and then business players remove stories with the equivalent estimate from the remaining plan and insert the new story.
- Re-estimate: If Development players feel that the plan no longer provides an accurate map of development, they can re-estimate all of the remaining stories and set velocity again.

Reflection Workshop (Cockburn, 2002a)

- *Purpose*: Review team practices in the past period or set desirable team practices for the upcoming period.
- *Description*: This workshop involves all team members meeting in the same room to discuss what practices are going well with the team and what practices should be improved or performed differently in the upcoming/next period. These reviews are written on a flipchart and posted in a prominently visible place so that team members are reminded about them during their work.

The flipchart may have the following columns:

1. Keep these: which lists the practices that have been used and are still desirable for the upcoming/next period (e.g., daily meetings, pair programming)
2. Problems: which lists the current problems in team practices (e.g., too many interruptions, shipping buggy code)
3. Try these: which lists the practices that have not been used but are desirable for the upcoming/next period (e.g., pair testing, fines for interruptions)

The workshop should be 1 to 2 hours long. At the start of the next workshop, the team should bring in the flipchart from the previous workshop and start by asking whether this way of writing and posting the workshop outcome was effective.

Crystal methodologies suggest that a team should hold reflection workshops at the beginning and end of each release, probably also mid-release (Cockburn, 2002a).

Role Rotation (Millis & Cottell, 1997; Soller, 2001; Adams, 2003):

- *Purpose*: Encourage participation of team members; balance workload amongst team members.
- *Description*: The roles assigned to team members can be rotated amongst these members during the course of a team's shared task or during the lifetime of the team. For example, the role of "quality controller" may be assigned to different team members in different tasks of the team.

Role rotation can form positive interdependence between team members. It discourages domination by one person—a problem common in less-structured teamwork—and gives all members an opportunity to experience and learn from the different positions.

It may be useful to give the rationale for role rotation practice to team members before they join the team.

Round-Robin Participation Technique (Jarboe, 1996)

- *Purpose*: Encourage participation of team members during a particular team's shared task (e.g., discussion, workshop, creation of a particular work product); balance workload amongst team members.
- *Description*: The round-robin technique establishes an environment in which each team member, in turn, has the opportunity to express themselves openly without their teammates interrupting or evaluating their opinions.

Simple Design

- *Purpose*: Help to produce fast, economic, easy-to-understand, and easy-to-modify designs in XP development.
- *Description*: "Simple design" technique is often known as "do the simplest thing that could possibly work" or "you are not going to need it" (YAGNI). It recommends two rules (Highsmith, 2000):

    1. Only design for the functionality that is required in the forthcoming iteration, not for potential future functionality.
    2. Create the simplest design that can deliver that functionality. Beck (2000) gives four criteria for a simple design:

        - Run all the tests;
        - Have no duplicated logic, including hidden duplication like parallel class hierarchies;
        - State every intention important to the programmers (i.e., the generated code should be easy to read and understand); and
        - Have the fewest possible classes and methods.

Small/Short Releases

- *Purpose*: Allow for fast delivery of products to customers, particularly in an agile development project.
- *Description*: One of the principles of agile development is to deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale (Agile Manifesto, 2001). Accordingly, the "small release" technique suggests that every release should be as small as possible, containing the most valuable business requirements (Beck, 2000; Abrahamsson, Salo, Ronkainen, & Warsta, 2002). The initial simple system should be in production rapidly. New versions are then released on a very short cycle, even daily, but at least monthly.

Note that a release has to make sense as a whole, that is, it cannot implement half a feature just to make the release cycle short.

In order to practice the "small release" technique, the developers should (Beck, 2000):

- Determine beforehand what are the most valuable features or business requirements, so even a small system would have business value.
- Integrate continuous (see technique "continuous integration"), so the cost of packaging a release is minimal.

Sprint/iteration Review Meeting

- *Purpose*: Present the final product of an iteration/sprint to management, customers, users, and product owner; review past performance of the sprint.
- *Description*: On the last day of an iteration/sprint, a sprint/iteration review meeting is

held and conducted by the Scrum master. In the meeting, the development team presents the product to management, customers, users and product owner, reporting the system architecture, design, functionality, strengths, and weaknesses.

The sprint goal and "sprint backlog document" are compared to the actual product of the sprint, and reasons for any discrepancies are discussed. The sprint/iteration review meeting may uncover new backlog items to be added to the "release backlog document" (and "product backlog document") and may even change the direction of the project. Difficulties and successes of the project team during the iteration/sprint should also be reported. All stakeholders can then make an informed decision on what to do next (i.e., next sprint or release).

The sprint/iteration review meeting should be conducted in an informal manner.

Sprint Planning Meeting

- *Purpose*: Assist planning for an iteration/sprint in Scrum (see "develop iteration plan" in TaskKind section).
- *Description*: A sprint planning meeting actually consists of two consecutive meetings organized by the "Scrum master." In the first meeting, the developers, product owner, Scrum master, customers, and management meet with each other to decide upon what functionality to build during the forthcoming iteration/sprint. In the second meeting, the developers work by themselves to figure out how they are going to build this functionality during the iteration/sprint.

Inputs to the sprint planning meeting are "release backlog documents," the most recent iteration product, business conditions, and the capabilities and past performance of the project team. Output of the meeting is a "sprint backlog document."

To start the first sub-meeting of the sprint planning meeting, the product owner presents the top priority items in the "release backlog document" to everyone and leads the discussion on what everyone wants the project team to work on next and what changes to the current "release backlog document" (and "product backlog document") are appropriate. Having selected the backlog items for the iteration/sprint, a sprint goal is drafted. A sprint goal is an objective that will be met through the implementation of the selected backlog items. This goal is needed to serve as a minimum, high-level success criteria for the sprint and keep the project team focused on the big picture, rather than just on the chosen functionality. If the work turns out to be harder than the team had expected, the team might only partially implement the functionality but should still keep the sprint goal in mind.

In the second sub-meeting, the developers compile the "sprint backlog document" by identifying a list of tasks that they have to complete during the iteration/sprint in order to reach the sprint goal. These tasks are the detailed pieces of work needed to convert the selected backlog items into working software. The developers have the total freedom and autonomy in determining these tasks. Each task should take roughly 4-16 hours to finish. The sprint backlog is normally modified throughout the iteration/sprint. All developers are required to be present at the second sub-meeting. The product owner often attends.

System Metaphor

- *Purpose*: Help to establish a shared understanding between everyone in the project team about the future system, including its structure and how it works. The metaphor in XP replaces much of the system architecture, by identifying the key objects and their interactions. This technique can be used to support task "describe application" (see TaskKind section).

- *Description*: A metaphor is a story that everyone—customers, programmers, and managers—can tell about how the system works (Beck, 2000). It provides an overall view of how the future system is perceived, while the "stories" that are used to describe individual features of the system. For example, a customer service support system can be described by the following potential metaphors:

- Naïve metaphor (where the real-life objects are referred to themselves): For example, customer service representatives create problem reports on behalf of customers and assign them to technicians.
- Assembly line metaphor: Problem reports and solutions are thought of as an assembly and the technicians and customer service representatives are workers at stations.
- Subcontractors metaphor: The customer service representatives are general contractors, with control over the whole job. They can let work out to subcontractors (i.e., technicians).
- Problem-solving chalkboard metaphor: The customer service representatives and technicians are experts who put a problem on the board and solve the problem.

In XP, the most appropriate metaphor of the system should be determined during the "exploration phase," when user stories are written (Wake, 2001). It can be revised over time as the project team members learn more about the system. The whole project team should agree on the key objects in the metaphor. The metaphor can be used to help orient the developers when they are trying to understand the system functionality at the highest levels, and to guide the developers' solutions. Use the object names in the metaphor as the "uppermost" classes in code. Methods, variables, and basic responsibilities of the system can also be derived from the metaphor.

Team Facilitation

- *Purpose:* Facilitation is "the art of leading people through processes towards agreed-upon objectives in a manner that encourages participation, ownership, and creativity from all involved" (Sibbet, 2002).
- *Description:* Generally, the processes and methods of facilitation provide value to teams by enabling the following (Paulsen, 2004): 1) shared responsibility for outcomes; 2) individual accountability; 3) improved decision-making, problem-solving, and conflict resolution; 4) staff participation and empowerment; 5) efficient use of resources; 6) creative teams and projects; 7) productive meetings and teams; 8) flexible responses to changes; 9) alignment to common plans or goals; and 10) organizational learning.

Basic facilitation techniques include verbal techniques and nonverbal techniques (Bloom, 2000; Rees, 2001).

- Verbal techniques: Asking questions, redirecting, referencing back, paraphrasing, humor, positive reinforcement, obtain examples, unity and diversity, and meta-communication and skill development.
- Nonverbal techniques: test inferences, active listening, voice, facial expressions, and silence.

There are also facilitation techniques for "expanding" and "narrowing" processes. The process of expanding includes generating ideas and gathering information while the process of narrowing entails comparing and evaluating information and making decisions (Kearny, 1995). Examples of expanding facilitation techniques (Kearny, 1995; Rees, 2001) are brainstorming, nominal group process, fishbone (Ishikawa) diagram, and mapping the territory. Some narrowing facilitation techniques are sorting by category (affinity diagram), N/3 (rank

order), polling the group, jury of peers, and good news/bad news.

Team Motivation

- *Purpose*: Motivate productivity and morality of team members during team's tasks.
- *Description*: A huge pool of techniques for team motivation can be found in the literature on organizational management or human resource management. Here we just list some example techniques.

Grazier (1998) suggests 54 ideas for the motivation of generic teams. Some of which are:

- Vision and mission (the team's goals and tasks should be in line with the members' wants and needs).
- Challenge (the tasks and responsibilities assigned to team members should pose a challenge to them).
- Growth (the tasks and responsibilities assigned to team members should encourage their personal growth).
- Recognition (frequent appraisal and recognition of team member's contribution).
- Communications (open, direct communications).
- Responsibility (responsibility should be assigned to members together with authority. Responsibility can be de-motivating if the consequences of error or failure are too great).

Other recommended techniques fall in the categories of fun, exercises, training, and involvement.

Couger and Smith (1992a, b) propose some motivational techniques for IS development teams in particular. According to their international surveys of different job types, IS staff have the highest growth need (i.e., the need to achieve and accomplish tasks) and lowest social need (i.e., the need to interact with others) of all the professions surveyed. Accordingly, Couger and Smith suggest the following techniques for motivating IS team members:

- **Skill variety:** The tasks and responsibilities assigned to team members should require a number of different skills and talents from the members, thus posing a challenge to them.
- **Task identity:** The task assigned to team members should be a "whole" and identifiable piece of work.
- **Task significance:** The task assigned to team members should be clearly scoped in term of the overall project goal.
- **Autonomy:** Team members should be provided substantial freedom, independence, and discretion. The team leader/task facilitator should focus on the delivery of the products and not on the specific approach used to achieve them.
- **Feedback:** Team members should be given formal feedback on their performance and deliverable's quality.

Given that many of the team members in IS projects have high growth-need and low social-need, the above techniques can reinforce productive behaviour. Personnel with a high need for growth will readily accept the excitement of challenging work. A person with a low social need will be comfortable receiving such work and being allowed to complete it with low people interaction.

Test Driven Development

- *Purpose*: Facilitate automated unit testing of code.
- *Description*: Programmers develop code through rapid iterations of the following steps (Beck, 2003):

1. Writing automated test cases
2. Running these unit test cases to ensure they fail (since there is no code to run yet)

3. Implementing code which should allow the unit test cases to pass
4. Re-running the unit test cases to ensure they now pass with the new code
5. Refactoring the implementation or test code, as necessary
6. Periodically re-running all the test cases in the code base to ensure the new code does not break any previously running test cases.

APPENDIX C. ACTIVITY KINDS

Team Management Activity Kind: is an activity of ensuring that every team in the project functions in an effective manner. The objectives of the team management activity kind include:

- To ensure that each team member is clear of his role, responsibilities, and position within the team.
- To ensure that team members collaborate smoothly, with no intra-team or inter-teams unsolvable conflicts
- To ensure that the goals of each team are achieved or exceeded.

The team management activity kind involves team leaders (and/or team members) performing the following tasks (cf. TaskKind section):

- "Choose project teams"
- "Specify team structure"
- "Specify team policies"
- "Identify project toles and responsibilities"
- "Manage shared artefacts"
- "Mediate/monitor the performance of team's tasks"