Chapter XVI
Some Method Fragments for Agile Software Development

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

The use of a situational method engineering approach to create agile methodologies is demonstrated. Although existing method bases are shown to be deficient, we take one of these (that of the OPEN Process Framework) and propose additional method fragments specific to agile methodologies. These are derived from a study of several of the existing agile methods, each fragment being created from the relevant powertype pattern as standardized in the Australian Standard methodology metamodel of AS 4651.

INTRODUCTION

It is increasingly recognized that a universally applicable methodology (a.k.a. method) for software (and systems) development is not possible (Brooks, 1987; Avison & Wood-Harper, 1991; Fitzgerald, Russo, & O'Kane, 2003). One way to approach this is to eschew all attempts to create and promote a single methodology but instead to create a repository (or methodbase: Saeki, Iguchi, Wen-yin, & Shinohara, 1993) containing a large number of method fragments gleaned from a study of other methodologies, an evaluation of best industry practice, and so forth. Situational methods (Kumar & Welke, 1992; Odell, 1995) are then constructed by a method engineer "bottom up" from these fragments in such a way that they are "tailored" to the process requirements of the industry in question. This is the method engineering (ME) or situational method engineering (SME) approach to methodologies.

A second thread of relevance is the increasing interest, both in academe and industry, of agile

methods—methodological approaches to software development that tend to the minimalistic, focus on people rather than documented processes, and react well to rapidly changing requirements (Abrahamsson, Warsta, Siponen, & Ronkainen, 2003; Turk, France, & Rumpe, 2005). However, as published and often as practiced, these agile methods themselves may be overly rigid. To make them more flexible and possess so-called "dual agility" (Henderson-Sellers & Serour, 2005), a method engineering approach can be applied to agile methods as well as more traditional software development approaches. To do so, it is incumbent upon the method engineers who provide the method bases to ensure that these repositories of method fragments contain adequate fragments from which a range of agile methods can indeed be constructed.

In this chapter, we hypothesize that an agile method can be created from method fragments, once those fragments have been identified and appropriately documented. Following an introduction to the general characteristics of agile software development, we then examine an underpinning metamodel (AS4651). We then identify and document method fragments that conform to this metamodel and that support a range of agile methods including XP, Crystal, Scrum, ASD, SDSM, and FDD. We thus propose the addition of these newly document fragments to one extensive ME repository, that 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 methodbase. An important part of any such research is the validation phase. This is described in the complementary chapter (Tran, Henderson-Sellers, & Hawryszkiewycz, 2007), where we (re-)create four agile methods from the fragments in the newly enhanced OPF methodbase.

GENERAL CHARACTERISTICS OF AGILE SOFTWARE DEVELOPMENT

Although each agile development methodology is distinct, they do share some common characteristics. 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.

Firstly, agile development emphasizes the relationship and communality of software developers, as opposed to institutionalized processes and development tools. Valuing people over processes allows for more creativity in solutions. In the existing agile practices, this value manifests itself in *close team relationships*, *close working environment arrangements*, and other *procedures boosting team spirit*. The importance of teamwork to agile development has been emphasized by agilists (Cockburn & Highsmith, 2001; Highsmith & Cockburn, 2001).

Secondly, an important objective of the software team is to continuously produce tested working software. It is argued that documentation, while valuable, takes time to write and maintain, and is less valuable than a working product. Some agile methodologies promote *prototyping* (e.g., ASD), while others encourage *building simple but completely functional products quickly* as possible (e.g., XP).

Thirdly, *customer involvement* is promoted in all agile methodologies. The relationship and cooperation between the developers and the clients are given the preference over strict contracts. The clients are encouraged to actively participate in the development effort.

Fourthly, the developers must be prepared to *make changes* in response to the emerging/changing needs during the development process. Any *plan must be lightweight* and easily modifiable. The "plan" might simply be a set of post-it notes on a whiteboard (e.g., as in Scrum: Schwaber, 1995).

Boehm (2002) presents a comparison (Table 1) between agile development and conventional process-oriented development (or plan-driven as he calls them). This comparison helps to highlight further characteristics of agile methodologies. While many method fragments were identified in the era of plan-driven methodologies, the atomic nature of these method fragments should mean that they are equally usable for the creation of agile methods. Indeed, this is amply demonstrated in empirical studies (Henderson-Sellers & Serour, 2005), which illustrated how several Sydney-based organizations have successfully created an agile situational method.

In our research project, we aim to identify method fragments for supporting agile development by examining:

- The characteristics of agile development described above; and
- The existing prominent agile methodologies, namely XP, Scrum, adaptive software development (ASD), dynamic systems development methodology (DSDM), Crystal methodologies, and feature driven development (FDD).

In this chapter, only method fragments extracted from the general agility characteristics and XP, Scrum, Crystal clear, and Crystal orange are listed.

THE UNDERPINNING METAMODEL AND AVAILABLE REPOSITORY

When method fragments are extracted from a methodology, they need to conform to some standard. Here, we ensure that they conform to an official Australian Standard, AS 4651 (Standards Australia, 2004)—a standard metamodel for development methodologies that has recently been "internationalized" through the normal ISO process resulting in the international standard ISO/IEC 24744 in 2007.

Both AS 4651 (as described here) and the newer ISO/IEC 24744 use two important architectural elements that are outlined here: powertypes and a multi-level architecture aligned with the information systems/business domain in which software development takes place (Note that in ISO/IEC 24744, some of the metaclass names are slightly different from those in AS 4651. Here we use the AS 4651 names).

The overall architecture is shown in Figure 1. The three layers reflect best practice and are organized to match the conceptual concerns of various subgroups within the software engineering community. People working on an endeavour (e.g., a specific software development project) (in the "endeavour layer") utilize methodologies, tools and so forth, which are all defined in the "method layer." This pair of layers is all the software development team is concerned with. However, a different pair of layers is of interest to methodologists, method engineers, and tool builders: the method layer together with

Table 1. Comparison of agile and plan-driven methods

Home-ground area	Agile methods	Plan-driven methods
Developers	Agile, knowledgeable, collocated, and collaborative	Plan-oriented; adequate skills; access to external knowledge
Customers	Dedicated, knowledgeable, collocated, collaborative, representative, and empowered	Access to knowledgeable, collaborative, representative, and empowered customers
Requirements	Largely emergent; rapid change	Knowable early; largely stable
Architecture	Designed for current requirements	Designed for current and foreseeable requirements
Refactoring	Inexpensive	Expensive
Size	Smaller teams and products	Larger teams and products
Primary objective	Rapid value	High assurance

the metamodel layer. It is this last (metamodel) layer that forms the basis and foundation for the others. This layer contains *all* the rule-focussed information necessary for creating methods, tools, and so forth.

The multilayer architecture reflects best *practice* and is no longer governed by the is-an-instance-of relationship as in the OMG's strict metamodelling hierarchy. Classes belong to the most "natural" layer as defined by the software engineering group of people most likely to be interested in their definition and usage. In particular, we wish to define (and standardize) certain abstract features of methodology elements in such a way that their subtypes can be defined in the method layer rather than the metamodel layer. We also wish to be able to allocate values to some attributes at the method layer while leaving other attributes without values until the endeavour layer, that is, for attributes to straddle *two* layers—not possible with current, traditional instantiation-based metalevel architectures (such as that employed by the OMG). To accomplish both these goals, we introduce the notion of a powertype (Odell, 1994)—the current most promising solution. A powertype is a class that has instances that are subtypes of another class (the partitioned class). Together the powertype class and the partitioned class form a *powertype pattern* (Henderson-Sellers & Gonzalez-Perez, 2005a, b).

A powertype pattern (as used in AS 4651) is shown in the metamodel layer of Figure 2. In this example, the powertype class is DocumentKind and the partitioned class is document. This means that there is a generalization relationship across layers, between (here) requirements specification document and its supertype Document as well as the more regular instantiation relationship (here between requirements specification document and the DocumentKind class in the metamodel layer). In other words, requirements specification document is concurrently an object (an instance of DocumentKind) and a class (a subtype of document). Such an entity was called by Atkinson (1998) a "clabject"—clabjects are an essential component of the powertype approach. In this example, as an instance of DocumentKind, requirements specification document has attribute values of name=requirements specification document and MustBeApproved=Yes. It also has attributes derived from its subtyping of document (title and version) that need to be given values at the endeavour level. To do this, they are first transmitted unchanged via the generalization relationship from document to requirements specification document. An object in the endeavour layer called, say, "MySystem" requirements specification, then instantiates requirements specification document (in the method layer), consequently allocating values to these attributes—here the values are shown (in Figure 2) as Title="MySystem" Requirements Specification and Version=1.1.

Figure 1. Schematic of the architecture underpinning AS 4651

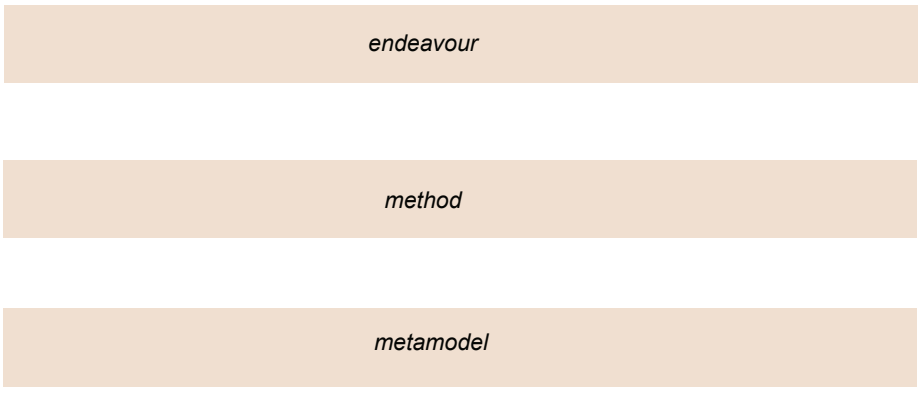
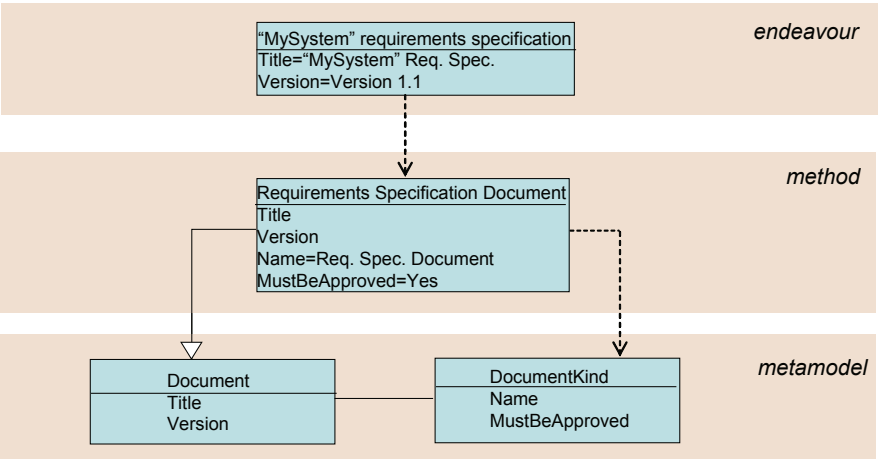


Figure 2. Powertype pattern showing how some attributes from the metamodel layer are instantiated at the model layer and others at the endeavour layer



Note that the suffix "Kind" is used to represent an element belonging to a *methodology*, which needs to be distinguished from an element that belongs to a particular *endeavour*. For example, "Producer" refers to people involved in a particular systems development project, while "ProducerKind" refer to kinds of producers described by the methodology used by that project. Note that project-level elements must be instances of some methodology-level elements. In this report, we only use *kind fragments, because we focus on the methodology level, not the project level.

The overall architecture of AS 4651 (and ISO/IEC 24744) is shown in Figure 3. Most of the classes in the metamodel participate in powertype patterns (left hand side) although some do not (right hand side). The instances of these latter classes are used at the method level as endeavour-independent sources of information rather than as classes from which instances can be created for a particular endeavour, for example, a programming language. These two categories were named templates and resources, respectively in Gonzalez-Perez and Henderson-Sellers (2005).

We will now describe in more detail each of the metaclasses that are relevant to agile fragments—the focus of this chapter.

Producer-Related Metaclasses

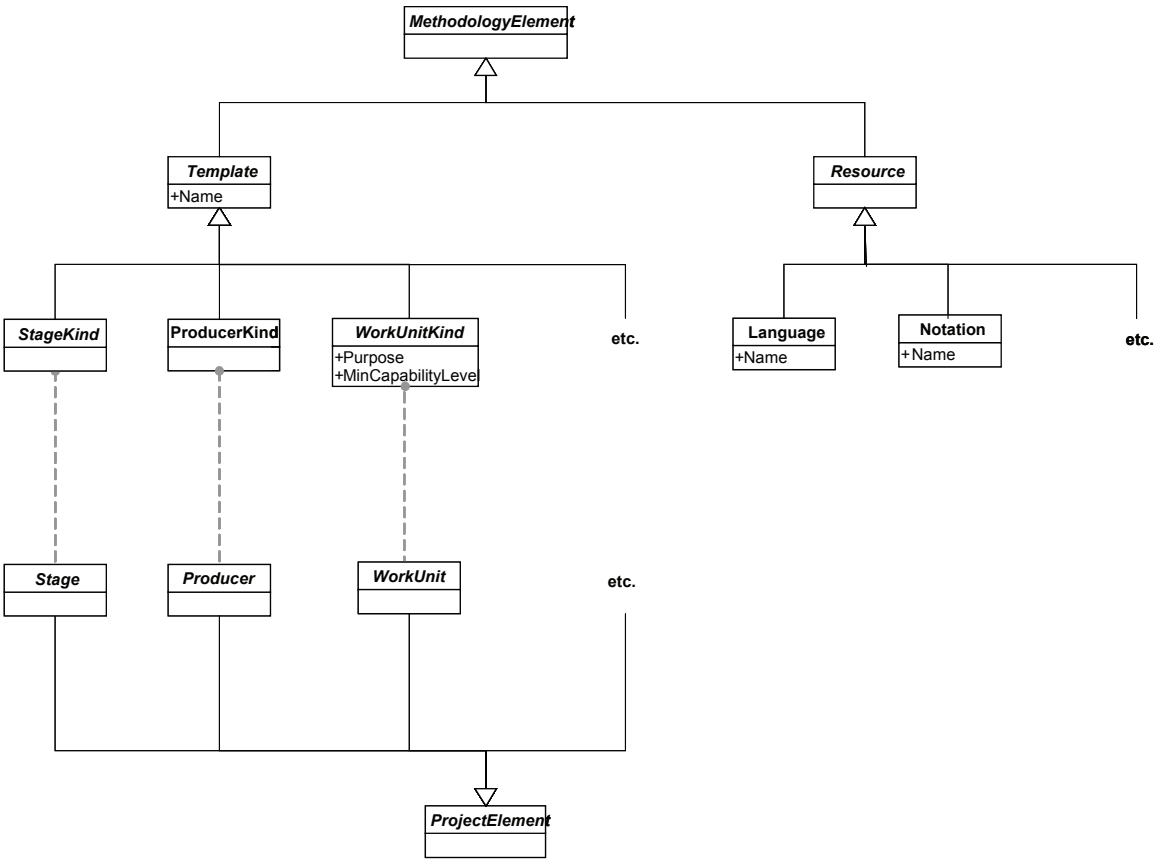
A producer is an agent that executes work units. A ProducerKind is a specific kind of producer, characterized by its area of expertise. The ProducerKind class is specialized into TeamKind, ToolKind, and RoleKind.

A team is an organized set of producers that collectively focus on common work units. A TeamKind is a specific kind of team, characterized by its responsibilities. A role is a collection of responsibilities that a producer can take. A RoleKind is a specific kind of role, characterized by the involved responsibilities. A tool is an instrument that allows another producer to perform a work unit in an automated way. A ToolKind is a specific kind of tool, characterized by its features.

WorkProduct-Related Metaclasses

A WorkProduct is an artefact of interest for the project. A WorkProductKind is a specific kind of

Figure 3. Overall architecture of AS 4651



work product, characterized by the nature of its contents and the intention behind its usage. It is specialized into DocumentKind and ModelKind. A document is a durable depiction of a fragment of the observed reality. A DocumentKind is a specific kind of document, characterized by its structure, type of content and purpose. It can contain other documents, recursively. In contrast, a model is a formal representation of some subject that acts as its surrogate for some well defined purpose. A ModelKind is a specific kind of model, characterized by its focus, purpose, and level of abstraction.

Although not directly needed in our current study, it is of interest to note that a ModelUnit is an atomic component of a model, representing a cohesive fragment of information in the subject modelled. A ModelUnitKind is a specific kind of model unit, characterized by the nature of the information it represents and the intention of using such representation. It allows a wide variety of subtypes; in particular, it supports the generation of all the metaclasses of a modeling language, assuming that modeling language definition has a "top" class equivalent to ModelUnitKind (e.g., the class element in UML Version 1.4 and 2.0)

Stage-Related Metaclasses

A Stage is a managed time frame within a project. A StageKind is 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.

WorkUnit- and Workflow-Related Metaclasses

There are two other groups of metaclasses of importance: WorkUnit and Workflow are the supertypes in question. Their definitions and the descriptions of agile method fragments conforming to these metaclasses will be discussed in a subsequent chapter of this book (Tran et al., 2007).

Discussion of OPF and its Repository

In order to capitalize on the use of method fragments, they need to be accumulated in a repository—here we utilize the repository of the OPEN Process Framework (OPF) (Henderson-Sellers & Graham, 1996; Firesmith & Henderson-Sellers, 2002), an example of an SME approach that uses the AS 4651 metamodelling approach. As well as the metamodel, the OPF also contains a well populated method fragment repository (see also http://www.opfro.org). The combination of the extensive methodbase content and the metamodel make OPF the best choice as the starting point for this investigation of agile method engineering.

The original work on the OPF was focussed on the necessary fragment support for object-oriented software development, although more recently it has been enhanced in order to support:

- Organizational transition (Henderson-Sellers & Serour, 2000; Serour, Henderson-Sellers, Hughes, Winder, & Chow, 2002)
- Web development (Haire, Henderson-Sellers, & Lowe, 2001; Henderson-Sellers, Haire, & Lowe, 2002)
- Component-based development (Henderson-Sellers, 2001)
- Agent-oriented development (Debenham & Henderson-Sellers, 2003; Henderson-Sellers, Giorgini, & Bresciani, 2004)
- Usage-centered design (Henderson-Sellers & Hutchison, 2003)
- Model transformations based on MDA (Pastor, Molina, & Henderson-Sellers, 2005)
- Aspect-oriented design (Henderson-Sellers, France, Georg, & Reddy, 2007)

Here, we first evaluate what current support is available for a range of agile methods. 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 Appendixes).

NEWLY IDENTIFIED FRAGMENTS TO SUPPORT AGILE DEVELOPMENT

This study has identified a large number of new fragments that could be considered for addition to the current OPF repository/method base. These are summarized in the following sections (and in Table 2) and are detailed in Appendices A-E in terms of the metaclass from which they are generated. Although listed in Table 2, those fragments in the context of WorkUnits and Workflows are not discussed here – details are to be found in the companion chapter (Tran et al., 2007).

Producer Fragments

There are three kinds of producer fragments: those from TeamKind, those derived from RoleKind, and those derived from ToolKind (These may also be constructed based on a set of coherent, identified responsibilities, since responsibility is an attribute of RoleKind).

TeamKind

Although there are three TeamKind fragments already in the OPF repository (peer programming team kind, XP-style team kind, and several subtypes of project team kind), our detailed analysis of XP, Scrum, and Crystal leads us to identify one further

Table 2. List of newly identified method fragments to support agile software development (N.B. There is no meaning to horizontal alignments).

ProducerKind Fragments	**WorkProductKind Fragments**	**WorkUnitKind Fragments**
Generated from TeamKind	*Generated from DocumentKind*	*Generated from TaskKind*
Scrum	Iteration plan	Design agile code
Generated from RoleKind	Product backlog	Develop release plan
Agile customer	Release plan	Explore architectural possibilities
Agile programmer	Story card	Manage shared artefacts
Coach	Team management (3 subtypes)	Mediate/monitor the performance of team's tasks
Consultant		Monitor work products
Product owner		Specify team policies
Scrum Master		Specify team structure
Tracker		Write user stories
XP tester	**StageKind Fragments**	
Generated from ToolKind	*Generated from StageWithDurationKind*	*Generated from TechniqueKind*
Groupware (6 subtypes)	Iteration/sprint	Agile team building
	Generated from InstantaneousStageKind	Collective ownership
	Iteration/sprint completed milestone	Conflict resolution
	Release completed milestone	Continuous integration
		Daily meeting
		Holistic diversity strategy
		Iteration planning game
		Methodology-tuning technique
		Monitoring by progress and stability
		Open workspace
		Pair programming
		Parallelism and flux
		Planning game
		Reflection workshop
		Role rotation
		Round-robin participation technique
		Simple design
		Small/short releases
		Sprint/iteration review meeting
		Sprint planning meeting
		System metaphor
		Team facilitation
		Team motivation
		Test driven development
		Generated from ActivityKind (sub-type of WorkFlowKind)
		Team management

missing fragment. Scrum has its own definition of Team, such that we must introduce a new fragment to represent this—we call is "Scrum Team Kind." A full description of this new Scrum Team Kind fragment is to be found in Appendix A.

RoleKind

The OPF repository contains already a large number of useful RoleKind fragments: programmer, peer programmer, customer, tester, project manager/big boss, several kinds of software engineers, and stakeholders such as user, manager, vendor representative, and of course customer.

Nevertheless, our detailed analysis of these three agile methods led us to identify eight new roles pertinent only to one or more of these agile approaches. These eight new roles are described in full in Appendix B.

ToolKind

To add to the two existing OPF fragments in this group, lowerCASE tool kind and upperCASE tool kind, we propose just one new one for furthering tool support for agile methods: Groupware ToolKind, which describes the kind of tools that support and augment group work (Greenberg, 1991). Their goal is to assist team members in communicating, collaborating and coordinating their activities (Ellis, Gibbs, & Rein, 1991). Groupware tools are particularly important in agile projects, where team members are required to work closely together and maintain a cohesive, mutually supportive team relationship.

Six potential sub-classes of Groupware Tool kind have been identified from the literature (Saunders, 1997; Terzis & Nixon, 1999) and are summarized in Appendix C.

This study suggests that future software development teams may begin to use Groupware based on the notions of agency, where the architecture of the Groupware consists of one or more agents (Tarumi, & Mizutani et al., 1999). This is still a subject of research.

Work Product Fragments

Work products can be classified as either documents or models, that is, instances of DocumentKind or ModelKind, respectively.

DocumentKind

An extensive list of document kinds has been documented in Firesmith and Henderson-Sellers (2002). These include build plans, system requirements specifications, user's annuals, templates, standards, design-focussed document sets, test sets, and documents relating to teamwork. To add to these, for the support of agile methodologies, we recommend five new document kinds, as described in Appendix D.

ModelKind

As discussed earlier, an agile project values "working software" more than documentation. Thus, except for user requirements that are documented by story cards, no formal models are required to capture analysis and design decisions. These decisions can be captured in the code itself. In other words, the code is the main repository of design information; formal models and diagrams are only developed if necessary, for example, to summarize and highlight important design issues at the end of the project (Fowler, 2001; Jeffries, 2004).

Stage Fragments

The OPF repository (Firesmith & Henderson-Sellers, 2002) already contains four useful StageWithDurationKinds (XP lifecycle, Scrum lifecycle, phase, and release build) and one useful InstantaneousStageKind (Code Drop Milestone)—these readily map to XP and Scrum phases (Table 3). In the former category, we suggest that agile methodologies need one further fragment (Iteration/Sprint BuildKind) and in the latter category it needs two (Iteration/Sprint Completed MilestoneKind, and

Table 3. Correspondence between OPEN phases and agile methodologies' phases

OPEN PhaseKinds	XP Phases	Scrum
Initiation	Exploration Planning	Pregame
Construction	Iterations to release	Development/Game
Delivery	Productionizing	Postgame
Usage	Maintenance Death	
Retirement		

ReleaseCompleted MilestoneKind) (for details see Appendix E).

SUMMARY, CONCLUSIONS, AND FURTHER WORK

We have argued that a situational method engineering approach can be used in the context of agile software development. Existing method bases have been shown to be deficient and in need of enhancement—in terms of more method fragments—in order to completely support these new methodologies. Based on a study of several of the existing agile methods, we have taken the existing methodbase of the OPEN Process Framework, or OPF (Firesmith & Henderson-Sellers, 2002), and proposed additions to it. These additions are a set of method fragments that uniquely support agile software development, each of which is created from the relevant powertype pattern as standardized in the Australian Standard methodology metamodel of AS 4651 (Standards Australia, 2004).

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., Warsta, J., Siponen, M. T., & Ronkainen, J. (2003). New directions on agile methods: a comparative analysis. In *Proceedings of ICSE '03* (pp. 244-254). Los Alamitos, CA, USA: IEEE Computer Society Press.

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

Atkinson, C. (1998). Supporting and applying the UML conceptual framework. In J. Bézivin & P.-A. Muller (Eds.), *«UML» 1998: Beyond the notation,* (Vol. 1618, pp. 21-36). Berlin, Germany: Springer-Verlag.

Avison, D. E., & Wood-Harper, A. T. (1991). Information systems development research: an exploration of ideas in practice. *The Computer Journal, 34*(2), 98-112.

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

Boehm, B. (2002). Get ready for agile methods, with care. *IEEE Computer, 35*(1), 64-69.

Brooks, F. P., Jr. (1987). No silver bullet: essence and accidents of software engineering. *IEEE Computer, 20*(4), 10-19.

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

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).*

Debenham, J., & Henderson-Sellers, B. (2003). Designing agent-based process systems—extending the OPEN Process Framework. In V. Plekhanova (Ed.), *Intelligent agent software engineering* (pp. 160-190). Hershey, PA, USA: Idea Group Publishing.

Ellis, C. A., Gibbs, S. J., & Rein, G. (1991). Groupware: Some issues and experiences. *Communications of the ACM, 34*(1), 39-58.

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

Fitzgerald, B., Russo, N. L., & O'Kane, T. (2003). Software development method tailoring at Motorola. *Communications of the ACM, 46*(4), 65-70.

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

Gonzalez-Perez, C., & Henderson-Sellers, B. (2005). Templates and resources in software development methodologies. *Journal of Object Technology, 4*(4), 173-190.

Greenberg, S. (1991). *Computer-supported co-operative work and Groupware.* London: Academic Press Ltd.

Haire, B., Henderson-Sellers, B., & Lowe, D. (2001). Supporting web development in the OPEN process: additional tasks. In *Proceedings of the 25th Annual International Computer Software and Applications Conference. COMPSAC 2001* (pp. 383-389). Los Alamitos, CA, USA: IEEE Computer Society Press.

Henderson-Sellers, B. (2001). An OPEN process for component-based development. In G.T. Heineman, & W. Councill (Eds.), *Component-based software engineering: Putting the pieces together* (pp. 321-340). Reading, MA, USA: Addison-Wesley.

Henderson-Sellers, B. (2006, May 30-31). Method engineering: theory and practice. In D. Karagiannis & H. C. Mayr (Eds.), *Proceedings of the Information Systems Technology and its Applications. 5th International Conference ISTA 2006.* Klagenfurt, Austria, **(**Vol. P-84, pp. 13-23). Bonn: Gesellschaft für Informatik.

Henderson-Sellers, B., France, R., Georg, G., & Reddy, R. (2007). A method engineering approach to developing aspect-oriented modelling processes based on the OPEN Process Framework. *Information and Software Technology, 49(7)*, 761-773

Henderson-Sellers, B., Giorgini, P., & Bresciani, P. (2004). Enhancing agent OPEN with concepts used in the tropos methodology. In A. Omicini, P. Pettra, & J. Pitt (Eds.), *Engineering Societies in the Agents World IV. 4th International Workshop, ESAW2003 LNAI* (Vol. 3071, pp 323-245). Berlin, Germany: Springer-Verlag.

Henderson-Sellers, B., & Gonzalez-Perez, C. (2005a). The rationale of powertype-based metamodelling to underpin software development methodologies. In S. Hartmann & M. Stumptner (Eds.), *Conferences in Research and Practice in Information Technology,* Sydney, NSW, (Vol. 43, pp. 7-16). Australia: Australian Computer Society.

Henderson-Sellers, B., & Gonzalez-Perez, C. (2005b). Connecting powertypes and stereotypes. *Journal of Object Technology, 4*(7), 83-96.

Henderson-Sellers, B., & Graham, I. M. (1996). OPEN: toward method convergence? *IEEE Computer, 29*(4), 86-89.

Henderson-Sellers, B., & Hutchison, J. (2003). Usage-centered design (UCD) and the OPEN Process Framework (OPF). In L. L. Constantine (Ed.), *Proceedings of the Performance by Design USE2003, Second International Conference on Us-*

age-Centered Design (pp. 171-196). Rowley, MA, USA: Ampersand Press.

Henderson-Sellers, B., & Serour, M. (2000). Creating a process for transitioning to object technology. In *Proceedings of the Seventh Asia--Pacific Software Engineering Conference APSEC 2000* (pp. 436-440). Los Alamitos, CA, USA: IEEE Computer Society Press.

Henderson-Sellers, B., & Serour, M. K. (2005). Creating a dual agility method - the value of method engineering. *Journal of Database Management, 16*(4), 1-24.

Highsmith, J., & Cockburn, A. (2001). Agile software development: the business of innovation. *IEEE Computer, 34*(9), 120-127.

Hogan, C. (2003). *The rules and practices of extreme programming*. Retrieved August 5, 2005, from http://www.everiware.com/cgi-bin/view/Lifecycle/ExtremeProgramming

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

Kumar, K., & Welke, R.J. (1992). Methodology engineering: a proposal for situation-specific methodology construction. In W. W. Cotterman, & J. A. Senn (Eds.), *Challenges and strategies for research in systems development* (pp. 257-269). Chichester, UK: John Wiley & Sons.

MountainGoatSoftware. (2005). *The Scrum development process*. Retrieved on August 23, 2005, from http://www.mountaingoatsoftware.com/Scrum/index.php

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

Odell, J. J. (1995). Introduction to method engineering. *Object Magazine, 5*(5).

Pastor, O., Molina, J. C., & Henderson-Sellers, B. (2005, May 27-June 1). Supporting ONME with a method engineering framework. *Proceedings of the Software Development. Int. Conf. on Software Development, SWDC-2005,* (pp. 195-208). Reykjavik, Iceland: University of Iceland Press.

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*.

Saeki, M., Iguchi, K., Wen-yin, K., & Shinohara, M. (1993). A meta-model for representing software specification & design methods. In *Proceedings of the IFIP WG8.1 Conference on Information Systems Development Process, Come* (pp. 149-166).

Saunders, J. H. (1997). *A manager's guide to computer supported collaborative work (also known as Groupware)*. Retrieved July 1, 2005, from http://www.johnsaunders.com/papers/cscw.htm

Schwaber, K. (1995). SCRUM development process. In *Proceedings of the OOPSLA'95 Workshop on Business Object Design and Implementation*.

Schwaber, K., & Beedle, M. (2002). *Agile software development with Scrum*. New Jersey, USA: Springer-Verlag.

Serour, M., Henderson-Sellers, B., Hughes, J., Winder, D., & Chow, L. (2002). Organizational transition to object technology: theory and practice. In Z. Bellahsène, D. Patel, & C. Rolland (Eds.), *Object-oriented information systems, LNCS 2425* (pp. 229-241). Berlin, Germany: Springer-Verlag.

Standards Australia. (2004). *Standard metamodel for software development methodologies* (AS 4651-2004). Sydney, NSW, Australia: Standards Australia. Purchasable online at http://www.saiglobal.com

Tarumi, H., Mizutani, S., et al. (1999). Simulation of agent-based Groupware with human factors. In *Proceedings of the 1999 International Symposium on Database Applications in non-traditional environments, Kyoto, Japan*.

Terzis, S., & Nixon, P. (1999). *Building the next generation Groupware: A survey of Groupware and its impact on the virtual enterpris* (Technical Report TCD-CS-1999-08). Dublin, Ireland: Trinity College, Computer Science Department.

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

Turk, D., France, R., & Rumpe, B. (2005). Assumptions underlying agile software-development processes. *Journal of Database Management, 16*(4), 62-87.

van Deursen, A. (2001). Customer involvement in extreme programming: XP2001 workshop report. *ACM SIGSOFT Software Engineering Notes, 26*(6), 70-73.

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

APPENDIX A. TEAM KINDS

Scrum Team Kind

This is a subclass of "Project Team Kind," which follows particular Scrum practices during the system development process. A Scrum team is different from an "XP-Style Team" in that its members can be cross-functional, including people with all of the skills necessary, for example, analysts, designers, quality control, and programmers (instead of only programmers as in an "XP-Style Team").

A Scrum team is characterized by its full authority to make any decisions and to do whatever is necessary to produce a product increment each sprint and to resolve problems/issues, being constrained only by organizational standards and conventions. The Scrum team should also self-organize to draw on its strengths and to allow everyone to contribute to the outcome. This need for self-organization implies that there should be no titles or job descriptions within a Scrum team. Each member applies his/her expertise to all of the problems. Scrum avoids people who refuse to code on the grounds that they are systems architects or designers.

APPENDIX B. ROLE KINDS

Agile Customer Role Kind

"Agile Customer RoleKind" is a subclass of "Customer RoleKind." Being a customer in an agile project requires many more responsibilities than a customer in a traditional development project. Traditional customers may only be involved at the inception of the project (e.g., helping to define requirements and contractual obligations) and at the end of the project (e.g., performing alpha, beta, and acceptance testing) (Coram & Bohner, 2005). In contrast, customers in agile projects are involved in the development process much more frequently and with more influence. In XP, at least one customer must be part of the project team and actively participate in the development process. Agile development style works best when customers operate in dedicated mode with the development team and when their tacit knowledge is sufficient for the full span of the application (Boehm, 2002). Note that merely having a customer representative available in the team is not sufficient. They must be committed, knowledgeable, collaborative, representative, and empowered (van Deursen, 2001; Boehm, 2002). An agile customer is required to be responsible for (and empowered to do) the following:

- Writing "stories" or listing "backlog items" to describe to developers the requirements of end users
- Making decisions in release planning and iteration planning (namely what requirements should be implemented in which release and which iteration, desired release date). This

involves making decisions on prioritizing and trading off the requirements
- Providing inputs (mainly opinions and decisions) into design and prototyping sessions
- Reviewing and accepting delivered releases
- Writing, organizing, and running functional tests on the delivered system. The customer will need to work closely with other project team members to learn what kind of things is helpful to test and what kind of tests are redundant
- Handling user training.

The best agile customers are those who will actually use the system being developed, but who also have a certain perspective on the problem to be solved (Beck, 2000).

Agile Programmer Role Kind

"Agile Programmer Role Kind" is a subclass of "Peer Programmer Role Kind." An agile programmer is responsible for not only the basic responsibilities of writing, unit testing, and debugging source code, but also responsible for:

- Analyzing user requirement
- Estimating how much effort and time are needed to satisfy each user requirement, thereafter letting the customer know about this estimate in order for them to make the decision on what to include in each release
- Designing the software solution
- Refactoring source code to keep the code as simple and definitive as possible
- Writing and running tests to demonstrate some vital aspect of the software
- Integrating new code to base-lined code and make sure the integrated product passes all Regression Tests
- Communicating and coordinating with other programmers and team members. If the programs run, but there is some vital component of communication left to be done, the job of the agile programmer is not yet over.

Coach Role Kind

A coach is responsible for the development process of the XP team as a whole. However, a "coach" is not to be equated with a team leader. While team leaders are often isolated geniuses making the important decisions on the project, the measure of a coach is how few technical decision he makes. A coach's job is to get everyone else in the team making good decisions. Responsibilities of a coach are:

- Understanding the practices and values of XP deeply, so as to guide other team members in following the XP approach (e.g., what alternative XP techniques might help the current set of problems, how other teams are using XP, what the ideas behind XP are, and how they relate to the current situation)
- Noticing when people are deviating from the team's process (e.g., programmers are skipping unit tests) and bringing this to the individuals' or team's attention
- Seeing long-term refactoring goals and encouraging small-scale refactorings to address parts of these goals
- Helping programmers with individual technical skills, such as testing, formatting, and refactoring
- Explaining the process to upper-level managers.

The role of coach usually diminishes as the team matures.

Consultant Role Kind

A consultant is not a part of an XP team. Rather, he/she is an external specialist whom the team seeks for technical help. Normally, an XP team does not need to consult a specialist, but from time to time the team needs deep technical knowledge. The responsibility of a consultant is to teach XP team members how to solve a particular problem that the team needs to solve. The consultant must

not solve the problem by themselves. Instead, one or two team members will sit with the consultant while he/she solves the problem.

Product Owner Role Kind

A Product owner is responsible for managing and controlling the "Product Backlog" in Scrum (see DocumentKind section). Their specific responsibilities are:

- Creating the product backlog together with the "Scrum Master" and project team members
- Maintaining and sustaining the content and priority of the product backlog, including adding, removing and updating product backlog items and their priority during releases and iterations/sprints. Note that the product owner solely controls the product backlog. Any member wanting to update/add/remove an item or its priority has to convince the product owner to make the change. Without a single product owner, floundering, contention and conflicts surrounding the product backlog result
- Turning 'issues' in product backlog into specific features or technology to be developed (i.e., workable items)
- Ensuring the product backlog is visible to everyone
- Segmenting and allocating product backlog items into probable releases, thereby developing the "Release Backlog Document" (see DocumentKind section)
- Working with project team members to estimating the amount of work in days to implement each product backlog item for "Product Backlog Document" and "Release Backlog Document" (see DocumentKind section)
- Revising the "Release Backlog Document" as the project team builds the product during each iteration/sprint (e.g., revising the release date or release functionality).
- Making final decisions on the tasks related to product backlog items, thereby developing "Sprint Backlog Document"
- Reviewing the system with other stakeholders at the end of iteration/sprint.

In a Scrum project, a product owner is chosen by the "Scrum Master," customers, and management.

Scrum Master Role Kind

Scrum introduces the role of "Scrum Master," which is essentially a sub-class of both "Coach Role Kind" and "Project Manager/big Boss Role Kind" in XP. A Scrum Master is a coach in that he/she is responsible for guiding the project team members in following the Scrum practices and values, for keeping track of the progress and ensuring everyone is on track, and providing assistance to members that need help (as well as for other responsibilities of a coach; see Coach Role Kind section). A Scrum Master is also a project manager in that he/she works with management to form the project team, represents the team and management to each other, and makes decisions. An important responsibility of a Scrum Master (which is not specified for a coach or a project manager) is to ensure that any impediments to the project are promptly removed and changed in the process, so as to keep the team working as productively as possible (Schwaber & Beedle, 2002). The Scrum Master can either personally remove them, or cause them to be removed as soon as possible. When the Scrum Master does the latter, he or she needs to make visible to the team a particular procedure, structure or facility that is hurting productivity. Another responsibility is to conduct the "Daily Scrum Meeting" and "Sprint/Iteration Review Meeting" (see TechniqueKind section in Tran et al., 2007).

Tracker Role Kind

The role kind "tracker" is introduced based on an XP tracker. A tracker is responsible for giving feedback to other members of an XP team. In particular, he/she handles the following responsibilities:

- Tracing the estimates made by the team (in release planning and iterative planning) and giving feedback on how accurate these estimates turn out, in order for the team to improve future estimations.
- Tracing the progress of each iteration and evaluating whether the team is able to achieve the desired goal if they follow the current course or if they need to change something. A couple of iterations into a release, a tracker should be able to tell the team whether they are going to make the next release without making big changes.
- Keeping a log of functional test scores, reported defects, who accepts responsibility for each of them, and what test cases were added on each defect's behalf.

XP Tester Role Kind

In an XP team, a lot of testing responsibilities actually lie with the "Agile Programmer Role Kind" (i.e., unit testing) and "Agile Customer Role Kind" (i.e., acceptance/functional testing). Thus, the responsibility of a tester role is really to help the customer write and execute functional tests. Accordingly, we introduce an "XP Tester Role Kind" as a subclass of "Tester Role Kind" who is responsible for helping the customer write and execute functional tests. An XP tester is also responsible for making sure the functional tests are run regularly and the test results are broadcasted in a prominent place.

APPENDIX C. SIX SUBTYPES OF GROUPWARE TOOLKIND

Conferencing Tool Kind

- Text-based conferencing: IRC, COW (conferencing on the Web)
- Audio/video conferencing: CUSeeMe, Sun Show Me, Intel TeamStation, PictureTel

Electronic Mail Tool Kind

- E-mail systems that support message-based collaboration and coordination: Lotus Notes, Novel Groupwise, and MS Exchange (these offer support for calendaring & scheduling, discussion groups, & notetaking)
- Newgroups systems: USENETS and GroupLens

Group Decision Support Tool Kind

- Support for group-agenda setting, brainstorming, filtering, classifying, or prioritizing the issues at hand: GroupSystems, MS NetMeeting, Meeting Room, TeamEC, ICBWorks

Meeting Support Tool Kind

- Support for audio-video conferencing and application-data sharing: MS NetMeeting, NewStar Sound IDEAS, and GroCo
- Support for the preparation and management of team meetings: DOLPHIN

Shared Workspace Tool Kind

- Sharedspaces, GMD FIT BSCW (basic support for cooperative work), Collaborative Virtual Workspace
- Room-based systems: TeamRooms, Mushroom
- Virtual environments: Virtual Society
- Support for group coordination: Lotus Notes, IBM FlowMark, JetForm, Action Workflow

Workflow Tool Kind

- Support for group coordination: Lotus Notes, IBM FlowMark, JetForm, Action Workflow

APPENDIX D. DOCUMENTKINDS

Iteration Plan Document Kind

- *Purpose*: A subclass of "Build Plan Document Kind," which documents the plan for a particular iteration/sprint within a release.
- *Description*: An "Iteration Plan Document Kind" specifies the *requirements* to be implemented in the forthcoming iteration/sprint, the *tasks* to be performed during the iteration/sprint to implement these requirements, and the *time estimates* to complete each task.

In XP, the requirements included in "Iteration Plan Document Kind" are 'user stories' selected from "Release Plan Document." The iteration plan is to be generated by XP programmers. These programmers also need to sign up for individual tasks and this information should also be recorded in the "Iteration Plan Document Kind" (Wake, 2001).

In Scrum, the "Iteration Plan Document Kind" is referred to as "Sprint Backlog Document." Requirements listed in it are backlog items selected from "Release Backlog Document" (Schwaber & Beedle, 2002). Once a task is started, its time estimate is to be updated daily (by the developer working on the task) to show the remaining hours needed to complete that work. Sprint backlogs are produced by the developers, "Scrum Master" and "Product Owner" (see RoleKind section).

Product Backlog Document Kind

- *Purpose*: A subclass of "System Requirements Specification Document Kind" generated and used in Scrum projects. Product backlog documents can be produced by multiple stakeholders, including customers, users, project team, marketing, sales division, customer support, and management.
- *Content*: A product backlog contains a master list of all requirements that can be foreseen for a system product. Product backlog items can include, for example, features, functions, bug fixes, defects, requested enhancements, technology upgrades, and issues requiring solution before other backlog items can be done (Schwaber & Beedle, 2002). These items can be technical (e.g., "refactor the login class to throw an exception") or more user-centric (e.g., "allow undo on the setup screen"). It is possible to express each Scrum backlog item in the form of XP's user story (see "Story Card Document Kind") (MountainGoatSoftware, 2005).

The list of product backlog items should be prioritized by the "Product Owner" (see RoleKind section). Items that have high priority are the ones that are the most desired. The effort needed for each item's implementation should also be estimated by the "Product Owner". The Product backlog is to be constantly expanded or updated with new and more detailed items, new priority order and more accurate estimations, as more is learned about the product and its customers (particularly throughout sprints and releases).

Release Plan Document Kind

- *Purpose*: A subclass of "Build Plan Document Kind," which documents the overall plan for a particular release.
- *Description*: A "Release Plan Document kind" specifies which requirements are going to be implemented by a particular release, the prioritization of these stories and the estimated date of the release (Wake, 2001; Hogan, 2003). A release plan will be used to create iteration plans (see "Iteration Plan Document Kind").

In XP, the requirements listed in the release plan are user stories selected from "story card documents." The release plan is to be developed by both development and business actors. A release plan used to be called "commitment schedule" in XP. The name was changed to more accurately

describe its purpose and be more consistent with "iteration plan" (Hogan, 2003).

In Scrum, the "Release Plan Document" is referred to as a "Release Backlog Document." It is to be developed by the "Product Owner" (see RoleKind section).

Story Card Document Kind

- *Purpose*: A subclass of "System Requirements Specification Document Kind" which is generated and used in XP projects. Story card documents are typically produced by customers in XP teams.
- *Content*: Each story card captures a "user story" describing a feature that the customer wants the system to provide. Each story is accompanied with a name and a short paragraph documenting the purpose of the story.

Team Management Document Kinds:

a. Team Structure Document Kind

- *Purpose*: This document kind is equivalent to the organization chart document kind, but at the team level.
- *Content*: This document kind should contain the specification of the structure of a particular team in terms of:
  - Roles (or individuals) that make up the team
  - Acquaintance relationships amongst these roles
  - Authority relationships that govern these acquaintances

The team structure document kind can be developed and updated by team leaders and distributed to newly joined team members.

b. Team Policies Document Kind

- *Purpose*: Specify team policies (or rules or conventions).
- *Content*: When working in teams, developers usually have to comply with certain policies (or rules or conventions) that govern the collaborative work within the team. These policies should be identified and documented. Example policies: each team member can only play one role at a time within the team; every team member must report to team leader; interactions/communications amongst team members are mediated by team leader.

The team structure document kind can be developed and updated by team leaders and distributed to team members.

c. Artefact Access Permissions Document Kind

- *Purpose*: Specify access permissions of particular artefact(s).
- *Content*: Different roles in a team, or different teams, may have different permissions to access the same artefact (for example, a team's message board can be read and updated by a team leader, but only read by team members). In such cases, the artefact should be accompanied by an "artefact access permission document," which specifies the permissions granted to each different role/team.

The artefact access permissions document kind can be produced and kept by the artefact manager role or team leader role (depending on which role is responsible for managing the artefact) and distributed to team members (probably only the permissions that the member is concerned).

APPENDIX E. BUILD KINDS

Iteration/Sprint Build Kind

An "Iteration/Sprint Build"[1] is a period of time from one to 4 weeks within a "Release Build" dur-

ing which a new set of features/requirements are implemented and added to a release. Each "Release Build" should be broken into several "Iteration/ Sprint Builds."

At the beginning of an iteration/sprint, "Develop Iteration Plan Task"[2] is performed to determine what features/requirements are to be implemented in that iteration/sprint. During the iteration/sprint, the project team designs, codes, and tests for the selected features/requirements. At the end of the iteration/sprint, various "Testing Tasks"[3] are carried out to see if the software produced by the iteration/ sprint satisfies the desired requirements.

In Scrum projects, each sprint also involves "Iteration Review" and "Iteration Adjust" tasks[4] which identify any risks/issues affecting the iteration/sprint and adjust the iteration/sprint or the overall requirements (or even development direction) to reconcile these risks/issues.

Iteration/Sprint Completed Milestone Kind

This milestone marks the event when an iteration/ sprint is completed. Ideally at each "Iteration Completed Milestone," the customer will have completed the functional tests on the resulting code and these tests should all pass. In XP, "Iteration Completed Milestones" occur during the XP "Iteration to First Release" phase, "Productionizing" phase, and "Maintenance" phase. The first ever iteration should put the overall system's architecture in place. In Scrum, "Sprint Completed Milestones" occur during Scrum's "Game" phase.

Release Completed Milestone Kind

This milestone marks the event when a release of the system is delivered to the customer. In Scrum, the whole Scrum's cycle (including "Pregame," "Game," and "Postgame" phases) works towards a particular release. Thus, the "Release Completed Milestone" occurs at the end of the cycle, or more specifically, the end of the "Postgame" phase. In XP, however, the first release is produced at the end of the "Productionizing" phase, while subsequent releases are delivered during the "Maintenance" phase. This gives rise to two subtypes of "Release Completed Milestone Kind":

- "First Release Completed Milestone Kind"
- "Subsequent Release Completed Milestone Kind"

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 and 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 XP uses the term "iteration" while Scrum uses "sprint."
2 See TaskKind section in Tran et al. (2007)
3 See TaskKind section in Tran et al. (2007)
4 See TaskKind section in Tran et al. (2007)