December 2002

OUTLINE OF AN AGILE INCREMENTAL IMPLEMENTATION METHODOLOGY FOR ENTERPRISE SYSTEMS

Michael Stender
Fraunhofer Institute for Industrial Engineering

OUTLINE OF AN AGILE INCREMENTAL IMPLEMENTATION METHODOLOGY FOR ENTERPRISE SYSTEMS

Michael Stender
Fraunhofer Institute for Industrial Engineering (IAO)
Michael.Stender@iao.fraunhofer.de

Abstract

This paper describes AI²M – Agile Incremental Implementation Methodology - an incremental implementation methodology for enterprise systems. AI²M integrates ideas from agile software development with the incremental implementation methodology Results Driven Incrementalism (RDI). An AI²M iteration is focused on improving a pre-defined business goal in a fixed time, opposed to implementing a certain functional module of an application package of conventional incremental implementation methodologies. The procedure model of AI²M, its aspired advantages and current challenges are discussed. By combining two incremental methodologies from different application fields, AI²M aspires to offer a holistic enterprise system implementation approach.

AI²M is currently under development. A tool for supporting AI²M will be developed in conjunction with a CRM system vendor.

Keywords: Agile software development, agile implementation, enterprise system implementation, CRM, SME

Introduction

Results from recent empirical research (e.g. Adam and O'Doherty 2000, Sumner 2000) strongly recommend using iterative, incremental methodologies for implementing enterprise systems. Enterprise systems in the context of this paper are defined as standard software packages, like ERP or CRM systems, e.g. from vendors like SAP or Siebel. The pre-dominant existing incremental implementation approaches by the major application vendors are centered on the idea of implementing one pre-defined functional module of a standard application after the other. These approaches put technical issues of the existing software architecture into the foreground for defining the goals of an iteration cycle. An example for a more user driven incremental implementation approach is Results Driven Incrementalism (RDI) described by Fichman and Moses (1999). Within RDI a business goal is the driving force for defining the goals of an iteration. While this is certainly beneficial from a user point of view, it leaves the question of how to solve the technical issues raised by using such an approach.

In software engineering, several iterative, incremental development methodologies have been developed. So called "agile" software development methodologies are the latest, intensely debated instances of these. In contrast to the creation of new software addressed by most agile approaches, enterprise system implementation deals with customizing pre-packaged software. Furthermore issues of organizational modeling (organizational structures, definition of business processes, etc.) are not addressed by agile software development methodologies. Nevertheless important close relations between these fields can be seen: Modern software development favors system development through "gluing" software components (component based software engineering; e.g. Szyperski 1998), hence reducing system development to customizing software components. Supplementing an incremental implementation methodology such as RDI with incremental software development methods therefore seems to be a natural fit. This paper presents an outline for a so called "agile" incremental enterprise system implementation methodology based on the combination of ideas from RDI with ideas from agile software development.

The paper is structured as follows: First a short overview on the rationale for incremental implementation methodologies is presented. Next an outline of the principles of agile development is given. The methodology eXtreme Programming (XP) is used

as an example for presenting core concepts of agile development methodologies. Parallels and deviations between XP practices and current implementation practices are discussed. Based on these findings an outline for an "agile" implementation methodology is given in the next section. The paper closes with an outlook on future research in this direction conducted in the context of a started research project.

Incremental Enterprise System Implementation

The decisive point to use enterprise systems is that implementing and using standard software, seen as customizing pre-packaged software solutions, should be much faster and more efficient than developing the required functionality from scratch. Secondly, implementing standard software promises to be an act of re-using best practice business procedures coded into the software. Experience reports from implementation projects frequently cite that especially these organizational issues have to be considered as more important than technological implementation hurdles (e.g. Sumner 2000).

The users' preference for business process re-use is limited to a certain degree, as strategic and competitive reasons require remaining "different" from competitors, which might use the same enterprise system (e.g. Davenport 2002). Accordingly almost no two enterprise system implementations are equal. This has lead to highly integrated, feature rich, but also very complex systems, whose implementation require knowledgeable customization experts. E.g. for the market leading ERP software SAP R/3, customization settings can be made in 16,000 tables; so called pre-defined "user-exits" define standardized extension points, where custom coding in the SAP programming language ABAP can be integrated into the software. The general adoption of E-Commerce is another complexity driver, as this typically requires integrating business processes across organizations and existing systems.

The increasing complexity of enterprise system implementation projects has lead to the recommendation to favor incremental implementation approaches over "big-bang" approaches. "Incremental" is pre-dominantly used in this context with the meaning of breaking up the implementation of a standard application into separate smaller cycles, which target implementing one functional module of the enterprise system after the other. The most important aspired advantage for this approach is the reduced project complexity of each project cycle. Complexity in this context must be seen in terms of technological (e.g. less modules) and organizational complexity (e.g. less users). Empirical evidence has shown, that implementation effort is highly related to organizational *and* technological complexity (Francalanci 2001; Adam and Doherty 2000). Parr and Shanks (2000a) further argue that a distinction only between "big-bang" and incremental, in the sense of module-by-module, implementation approaches is nowadays too coarse grained, as most implementations are done on a module-by-module base; they propose a taxonomy of ERP incremental implementation approaches, distinguishing between "plain vanilla" (no customization), "middle road" (minor customizations) and "comprehensive" implementation approaches (extensive customizations). In summary, empirical evidence supports the idea of reducing project complexity either by using aspired functionality or technological complexity as a success factor for system implementations.

Opposed to the described system module centered definition of the term "incremental", Moses and Fichman propose with Results Drive Incrementalism (RDI) a contrary approach for complexity reduction (Fichman and Moses 2000). RDI focuses on implementing sets of functionality which aspire to achieve an improvement for a pre-agreed business goal for the iteration cycle. The business goal is the driving factor for the following implementation activities, which can mean to implement functionality from several functional modules of an enterprise system. This difference in focus can lead to quite contrary implementation goals for an iteration, but is more business-oriented as the conventional incremental approach. A cycle in RDI includes definition of business processes with the system, technological system implementation and organizational implementation. Fichman and Moses see the following points as the major advantages of their approach compared to conventional approaches:

- *Project steering and success measure*: As an iteration cycle has a clear business goal as a target (e.g. "reduced lead-times from order entry to delivery") it is more target focused. The business benefit of an iteration is obvious for the project team and validation of achieved results is far easier.

- *Economic advantages*: At the end of an implementation cycle an organization immediately receives business value through using the system. Accordingly an earlier payback of the implementation costs is expected, which maximizes the return on investment.

- *Organizational learning*: The holistic approach around a pre-defined business goal helps the organization to collectively identify the advantages and potentials of the new system and of the own organization. This builds on theories from

organizational learning, that best results in learning can be achieved through active practice. Evolutionary learning furthermore helps in expressing better qualified requirements for later iterations, as these are based on practical experience and knowledge about the system.

To implement RDI, Fichman and Moses consider technological and organizational divisibility of an enterprise system of primary importance. In their opinion current standard ERP applications are too monolithic for RDI. This opinion is supported from a technical viewpoint by Sprott (2000), who reports on the limitations of componentizing current enterprise systems.

In summary while there is a general recommendation to use incremental implementation, these are mostly focused on using structural elements of the system (modules) to structure the project. There is a lack of incremental approaches which aim to use user requirements as a guideline to structure the implementation project, due to a lack of functional and technological divisibility of current systems. In the following an incremental software development methodology is presented, "agile software development", which seems to be promising to supplement an incremental implementation approach as RDI for the required technological tasks.

Agile Software Development

Intention of Agile Development Methodologies

Agile software development methodologies can be characterized as iterative, incremental development methodologies, which combine a set of quality assurance practices, with an approach for high customer/user involvement and with recommendations for the working conditions of the development team. Agile methods basic underlying assumption is that the holistic combination of these methods as a whole challenges the general rule of software quality assurance, that requirement changes in late stages of a development cycle induce exponential costs to correct compared to earlier corrective actions. Agile methodologies claim that the combination of the set of methodologies allows keeping a linear correlation for corrective costs between the point in time of change of a requirement and its incorporation into the development. Furthermore agile development methodologies assume that constant change of requirements has to be considered as the rule rather than an exception, as users or customers are in most cases not able to express their full requirements in the beginning. The most important reason for this are dynamically changing business environments, which make requirements expressed at early stage obsolete in favor of other requirements during the running project. The "agile manifesto" (Agile Alliance 2001) summarizes the core of agile methods as agile methods favor persons and interactions over processes and tools, prefer working software over comprehensive documentation, seek customer collaboration in favor of contract negotiation and aim to respond to change in favor of following a pre-defined plan. Experience in the field has proven agile software development approaches valuable (e.g. Grenning 2001, Schuh 2001, Poole and Huisman 2001). The origins of agile development have to be seen in older iterative approaches, such as Boehm's spiral model (2000a, 2000b).

Characteristics of XP

The most intensely discussed agile software development methodology is eXtreme Programming (XP), presented by Beck (2000). There are other development methodologies, which are based on the same principles (e.g. SCRUM, Adaptive Software Development/Crystal and Dynamic System Development Method).

XP aspires its goals by employing a set of core methods (for a detailed overview see Beck (2000)); a selection of these are:

- **Incremental delivery:** The overall structure of a XP project is to develop a system in small functional releases, in relatively short intervals. This also includes the first system release, which should be released rather after a few weeks than months. Even if this first version does only offer small functionality, it should clarify the general direction of the project and allow a first evaluation.

- **Planning game:** At the beginning of an iteration cycle, the system users and developers should agree on the most valuable functionality to be developed. Functionality is described on story cards (comparable to use-cases (e.g. Schneider and Winter 1998). The "gaming" character is that the users decide out of a set of story cards, which stories shall be implemented, while the developers "price" the selected stories. At the end, the set of the most valuable stories for the users should be the result, which can be implemented by the developers in an agreed time frame with the available resources.

- **Pair programming:** All programming is done by pairs of programmers; pair programming should encourage communication within the project team and enhance code quality.

- **Continuous re-design (refactoring)**: The general design guideline is to keep the overall system design as simple as possible. Simple in this context is defined by concrete metrics for object oriented systems (e.g. no duplicate code, fewest possible classes). Opposed to designing flexibility into a system right from the beginning, the assumption is that requirements change anyway, so that the extra effort put into flexibility in the beginning are not worth the effort in that moment. This also implies that in a case of new requirements, the team should not hesitate to re-design the whole system if that allows an easier requirements implementation.

- **Code standards:** To allow continuous re-design and as a measure of quality assurance, coding standards are to be enforced.

- **Minimal documentation**: The source code is considered as the primary source of documentation; further documentation such as visual class model are accepted as tools during, but are not considered as worth saving.

- **Automated testing:** Automated testing is the core concept of XP to identify as early as possible problems in the current design.

- **Test case definition before coding:** The explicit definition of test cases in advance to system customization has proven valuable in the field of software development, as developers avoid the pitfall of involuntarily designing test cases suiting their code, rather than designing tests suiting the user requirements.

- **User integration:** The development team should either have a customer on site or work at the customers' site, to have short feedback cycles.

Some supporters and (many) opponents of agile development methodologies consider these as loosely structured methods and as orthogonal to processes established through software process improvement programs, which are often conducted under the light of certification (e.g. CMM, SPICE). Paulk (2001) analyzes the relation of XP to CMM practices and shows that XP practices clearly relate to above than industry average CMM levels.

The high user orientation in combination with the incremental approach makes agile software development an interesting base for filling the aforementioned gap of addressing technological divisibility in system customization. The hypothesis of considering XP practices as of value for system implementation is assumed. To identify areas where current practices differ from XP practices, XP practices are put into perspective of current implementation practices in the following.

Reflection of XP Practices to current Implementation Practices

Table 1 gives a preliminary overview on the difference of XP practices and current implementation practice. The comments are based on experience reports from literature, interviews with three ERP implementation consultants for the ERP system SAP R/3 and author's own experiences with ERP and CRM implementations at nine user organizations.

Based on these identified differences, an outline for a new implementation methodology named "AI²M" – agile incremental implementation methodology – is presented.

Outline of AI²M - Agile Incremental Implementation Methodology

AI²M Goal and Application Domain

AI²Ms primary goal is to provide an incremental implementation approach, which allows users to prioritize which functionality should be implemented first, based on their business priorities. An AI²M iteration cycle shall be limited to a fixed time, with preferences for rather reducing implemented functionality than extending the cycle time. AI²M is based on concepts of RDI and agile software development methodologies.

Table 1. Current Adoption of Agile Practices in Implementation Projects

Agile Practice	Current Adoption	
Incremental Rollout	Partly applied	The meaning of incremental in agile development is clearly closer to the meaning of incremental in the sense of RDI, than in the sense of a module-by-module implementation approach. Business priorities as in the sense of RDI are set in the "Planning Game" practice (see below). Furthermore current systems are rather unforgiving with wrong customizations settings: Once activated, it is quite difficult to transform to another possible solution. This has lead to minimum first iteration cycles of at least 6 months. Once a basic installation is running, minor changes are deployed on demand.
Planning Game	Not applied	The dominating planning approach from the user perspective is based on defining delivered system functionality (equaling modules). In response to this customer request, the implementation consultant estimates the required resources and time. It is usually more accepted to delay a system introduction, than to reduce functionality.
Pair Programming	Not applied	Depending on the type of implementation project according to Parr and Shanks (2000a) no coding is required ("plain vanilla"), minor coding is required ("middle road") or extensive coding is necessary ("comprehensive"). To the author's and interviewees' knowledge, pair programming is not used in the field. Considered of more importance than coding is to set the right customization settings; this requires extensive know-how in the case of SAP. As one interviewee puts it: "I spent most of the time searching for the right settings."
Continuous Re-Design	Not applied	The major application packages are considered as so complex, that no one fully understands the complete system. Despite ongoing re-engineering efforts, the major application packages are consisting of monolithic modules, which make it prohibitious for outsiders to re-design the application architecture (Sprott 2000). Customization is limited to the pre-defined options. If a customer needs functionality not foreseen through customization and minimal extra coding, it is recommended to contact the vendor.
Coding Standards	Partly Applied	Depends on local practices.
Minimal Documentation	Applied	The typical customer does not pay for documentation. Furthermore the applied customization settings are considered as a valuable know-how asset by the consultant implementing the application.
Test case design before coding (customization)	Not applied	While requirements for custom coding sometimes are available in written form, is not current practice to create test cases from requirements specifications. Small customization requirements are negotiated verbally, while for larger customization requirements written documentation is produced. Nevertheless it is not usual to derive test cases from these specifications in advance to customization or coding.
Automated Testing	Partly Applied	While seldom applied directly during customization, e.g. SAP recommends working with dedicated test systems, where customization settings can be tested in isolation to development and production systems. Automated procedures exist to transfer customization settings from development systems to test systems.
User Integration	Applied	Customization experts usually work at the customer site on customer machines. A high user orientation of customization work is the norm.

As outlined by Fichman and Moses (1999), not all application areas and software systems are equally suitable for an incremental implementation approach. There can be legal requirements prohibiting to change certain business practices over several iterations, e.g. as in accounting where it might be required to stay with one accounting practice from a certain key date. Other reasons include the above mentioned technological divisibility of the system to be implemented. To take the different technological levels of implementation projects into account, AI²M relies to different degrees on agile practices, according to the taxonomy of ERP implementation types ("plain-vanilla", "middle-road", "comprehensive") of Shanks and Parr (2000a). While legal constraints certainly have to be considered as "hard" constraints prohibiting an incremental approach, technical reasons are by nature more a shortcoming of the underlying systems architecture than a reason for dropping incremental approaches in general.

The primary target domain when designing AI²M was the implementation of Customer Relationship Management (CRM) systems for small and medium sized enterprises (SME). CRM systems address the support of business processes at the organizational interface between a business and its customers. This domain is considered as suitable for an incremental approach as on the one hand currently an increasing demand is foreseen in the field and on the other hand no "hard" constraints (e.g. legal requirements) prohibit enterprises from adopting an incremental approach. While the general focus of AI²M is on CRM systems, the following presentation is of a general nature, not limiting its applicability to the CRM domain.

First the general procedure model of AI²M is presented. The detailed structure of an AI²M iteration cycle including the rational for the used agile elements is described afterwards. The section closes with anticipated challenges for using AI²M.

AI²M Procedure Model

The overall flow of the AI²M approach is depicted in Figure 1: As characteristic for incremental approaches, the whole project is cut into smaller release cycles. As suggested by RDI, a release cycle is dedicated to improve a pre-agreed business goal. The definition of the business goal and the identification and scheduling of relevant functional modules of the system is part of the planning game, where implementation consultants and users negotiate the goals to be achieved.

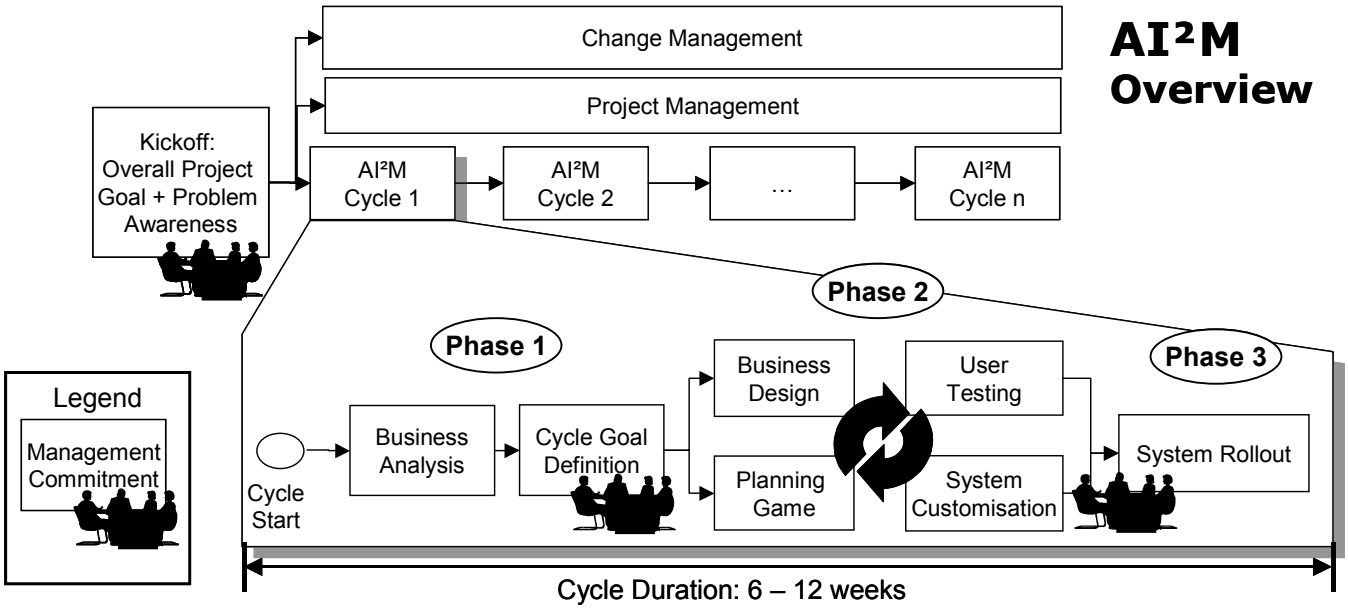

Figure 1. AI²M Project Structure Overview

The three main phases are characterized as follows:

- **Phase 1: Analysis**: Within analysis an overall business goal committed from top-management of the user organization shall be identified, which should be improved through the results of the iteration cycle. It has to be stressed, that this follows the idea of RDI, to focus on business goals instead of delivery of system functionality. An example for a business goal is "decreased lead time from order entry to order delivery".

- **Phase 2: Process design & System Customization:** Based on the aspired business goal, a set of relevant processes for this business goal shall be identified, which are to be supported by the enterprise system. The first process design phase shall be finished by a planning game, in which the process design and customization team agree on a set of functionality to be delivered in the iteration cycle. As outlined in the rules for the planning game in XP, the users are free to choose the functional scope of the next iteration, out of the possibilities constrained by the iteration timeline, available resources and quality consideration represented by the customization experts. This feedback shall help in prioritizing processes to be implemented in the next task. Processes to be implemented shall be documented as lightweight as possible; this means tasks,

sequence of tasks and roles need to be made clear, but the documentation should limit itself to the process to be implemented. To ease the process design phase, a system vendor could offer pre-configured process templates.

Based on the process descriptions, users and system experts formulate test cases *in advance to technical implementation* for testing the correctness from a business perspective of a customization. Test cases should be able to derive from the aspired user processes. Starting with a first set of test cases, the customization and implementation settings shall be made. "Pair customization" in analogy to pair programming should be aspired, as experience has shown the value of consistency and quality of achieved results. Depending on the tasks at hand, mixed user / system expert teams are recommended for easy tasks, also as a training measure. After completing a customization, automated testing shall verify the achieved result. If automated testing proves a desired level of quality, user testing shall verify the implementation from a user perspective. This phase shall finish with a commitment to put the achieved results into production. While this is the overall linear flow of tasks in this phase, the design foresees to go back into the task chain at any time if necessary. Most importantly in this context is the general rule, to *keep the planned delivery date of the iteration fixed and adapt other variables accordingly if required*. This means, that if during system customization it becomes clear, that not all user processes can be implemented, the functional scope of this iteration is cut, so that the delivery date can be kept. Furthermore if it gets clear that the test coverage of the current test set is insufficient, new test cases shall be added and integrated into the testing environment.

- **Phase 3: System rollout:** After completing the system customization, the system release shall be rolled out in the field. Roll-out shall be accomplished by typical measures, such as training and coaching.

After finishing such an iteration cycle, the next iteration cycle shall be started. It is aspired to achieve an iteration duration of 6 to 12 weeks. This has to be seen in the context of the target domain CRM systems and the target user group of SMEs. Obviously large user organizations might require other time scales, but probably not too much longer than three months, to avoid loosing the advantages of incremental delivery.

Structure of an AI²M Iteration Cycle

The detailed procedure model for an iteration cycle within AI²M is depicted in Figure 2: It describes the tasks, an implicit role model for a project team within a project cycle and the tangible result of each task. While there is a general hierarchically step-by-step task flow in a cycle, within each phase the tasks shall be conducted iteratively. Nevertheless the borders between the phases mark milestones, at which clear decisions need to be made, to enter the next phase. The integrated agile practices are marked in italics in the figure.

Table 2 gives an overview on the integrated concepts of agile software development into AI²M. Their use is differentiated according to the different implementation project types.

Table 2. Use of Agile Practices in AI²M

Agile Practice	AI²M Implementation Type			*Current* ***Adoption*** ***(see Table 1)***
	Plain Vanilla	Middle Road	Compre-hensive	
Incremental Rollout	+	+	+	*Partly applied*
Planning Game	+	+	+	*Not applied*
Pair Programming	-	O	+	*Not applied*
Continuous Re-Design	-	-	-	*Not applied*
Coding Standards	-	O	+	*Partly applied*
Design of test cases before customization / coding	+	+	+	*Not applied*
Minimal Documentation	+	+	+	*Applied*
Automated Testing	-	O	+	*Partly applied*
User Integration	+	+	+	*Applied*

= Use not foreseen, O = Decide on use on case-by-case basis, + = Use

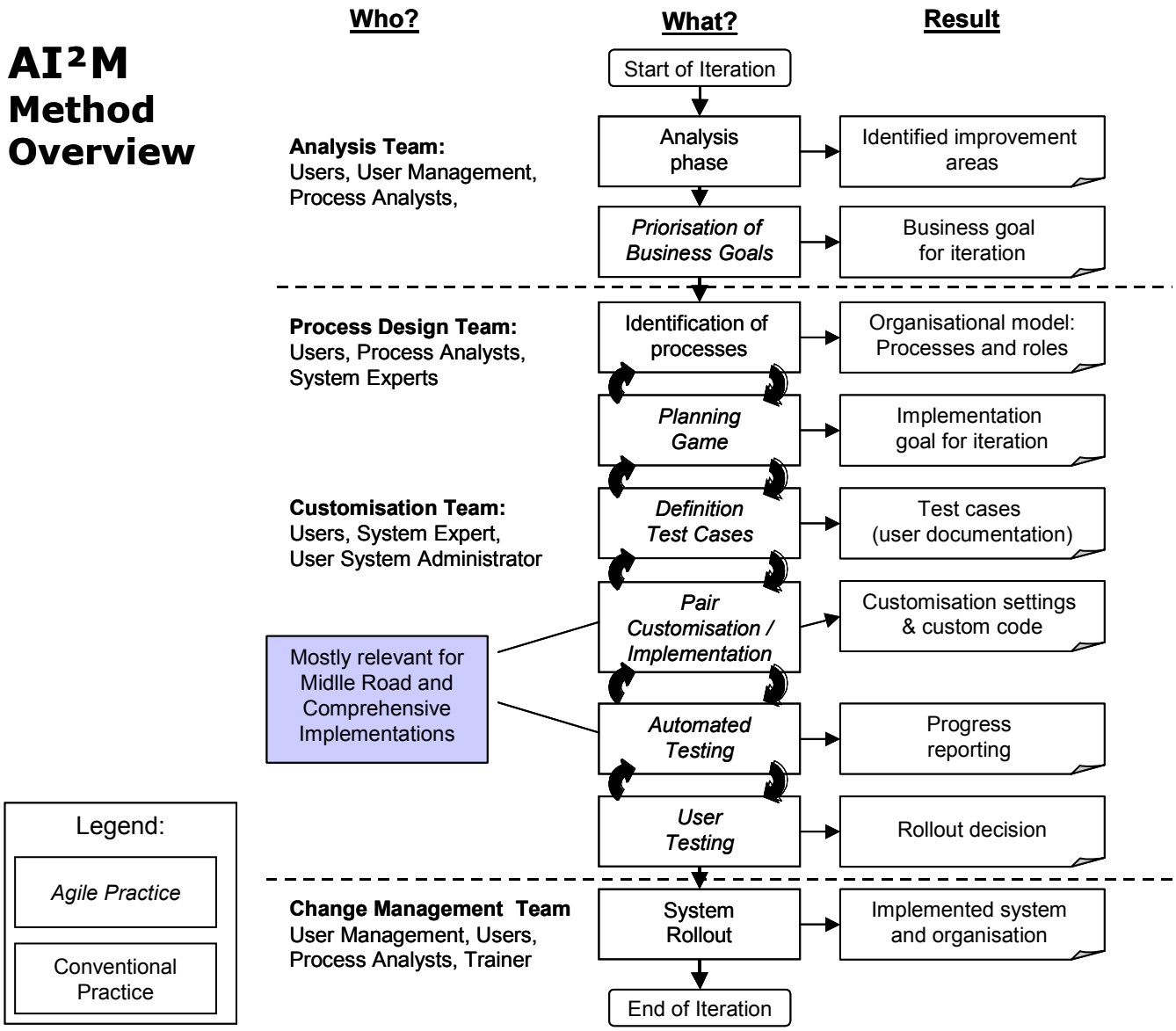

Figure 2. Process Model for an AI²M Iteration Cycle

The aspired advantages by including the different agile practices in AI²M are as follows:

- *Incremental delivery*: An AI²M cycle shall follow the idea of RDI, to focus on improving a business goal as the iteration goal. It follows the reasoning from RDI, that the users should enter an organization learning process to express better requirements. Additionally this approach is better able to cope with changing business needs, which might require giving an implementation project a new direction halfway through the project.

- *Planning Game*: At the beginning of an iteration, after a short analysis phase, a planning game shall be conducted. The planning game shall be the negotiating mechanism between user stakeholders and implementation experts for agreeing on concrete implementation goals for the iteration cycle. As in agile software development, an iteration shall focus on achieving the committed timetable and rather adapt the aspired iteration goals than changing the deadline.

- *Pair programming / customization*: AI²M shall evaluate the use of the instrument "pair customization". This shall address the problem, that system customization for complex systems is considered as kind of a "black magic" for certain systems (e.g. SAP R/3), where highly specialized experts are required. Teaming these activities has to be seen as a quality assurance activity and furthermore as a knowledge sharing and building activity, which even might be done as a training measure in conjunction with representatives from the user organization. According to the definition of plain-vanilla implementations pair customization/programming should only be of limited use.

- *Continuous re-design*: The complexity of existing large standard application prohibit a re-design for implementation consultants, as the disadvantages (loosing technical support and version compatibility) outweigh the advantages. It is preferable to stay with current practices in this area and to negotiate directly with the vendor in case of a need for fundamental architectural changes.

- *Coding standards*: If customized coding is required to a larger extent, coding standards should be enforced to ease intra- and inter project communication.

- *Design of test cases before customization:* Compared to agile development, test cases in AI²M should have a broader focus, being not limited to a technical test case. Instead a test case should consist of a description of an instance of a complete business process or use-case with sample data. Ideally a test case is a combination of sample data with system user documentation, with the users describing input and desired output, while the implementation partner describes the required functionality for the requested result. After completing system customization, test cases together with the process definitions shall form the basis for the user documentation. This shall help to keep documentation requirements to a minimum. It is desirable to achieve this through automated measures. Another use is for settling legal differences, as the test cases should be sufficiently clear.

- *Minimal documentation*: The production of user documentation shall be achieved on the basis of test cases (see above). For achieving minimal system documentation, tool support seems to be the most realistic choice.

- *Automated testing:* While having a specification or test cases at all is a pre-requisite for checking an implementation, it is the automization of testing that allows an early problem identification. Intense use of automated tests allows to easily identify correlations within the code when modifying parts of the system. Furthermore test reports provide an easy, less-effort, automatically generated overall view on the current status of system customization. Putting test results on a timeline gives an indicator on the overall progress of system customization. While automated testing is certainly valuable for comprehensive and middle-road implementations, its value is questionable in the case of plain-vanilla implementations.

In summary the general rule for the adoption of agile practices in AI²M is, that the closer the implementation project is to a software development project, in other words an implementation of type "comprehensive", the more concepts of agile development shall be integrated. Nevertheless even in plain-vanilla type implementations reasonable agile characteristics shall be integrated compared to current practices.

Implications for Research

While the overall approach seems to be promising, the following challenges need to be addressed before achieving the aspired advantages:

- While there is to a certain degree agreement that agile development practices are appropriate for small to medium sized development projects, it is currently unclear to what size these procedures scale up. Beck relates project size to development team size; it has to be questioned, how this would translate to the case of implementation projects and what an acceptable critical upper limit in terms of functional complexity, user numbers, geographically dispersed sites, etc. would be.

- While the concept of model driven implementation, e.g. as in the case of ARIS and SAP (Kirchmer 1999), has gained recognition, there is currently a lack of a lightweight method which allows the automatic conversion of process specifications first to test cases and then to user documentation.

- Current standard test tools for enterprise systems (if existent) are often oriented towards playback mechanisms of user input (e.g. keystrokes), but do not allow some kind of business semantics as an input.

- Existing enterprise system architectures are mostly monolithic, so that the required functional and technological divisibility is not given. On the other hand it is currently unclear what the right level of granularity would be preferable and how a finer granularity could be technically achieved. While component based software engineering seems to be promising in this context, there is a lack of appropriate standards for the integration of more complex components for building enterprise systems.

- An incremental implementation requires an evolution of functionalities and data of an enterprise system from one iteration cycle to the next. Appropriate procedures for ensuring data and system consistency, e.g. as in the case of upgrading several software components of an enterprise systems, are not well understood.

Implications for Practice

From a commercial point of view, the promise of a fixed set of functionality for a fixed price, as often requested by user organizations, is complex in an incremental implementation scenario, as the business goals evolve over time. Accordingly it can be expected that the contractual situation between the implementing partners cannot rely on strict goals defined in advance, meaning less project control. On the other hand, the iterations with the planning game offer shorter feedback cycles, easing project steering and therefore strengthening project control.

The adoption of an incremental implementation approach has fundamental implications for users in the field: First enterprise system users need to have clear ideas about their business priorities in the course of a project. This requires a profound reflection of the current situation and the targeted strategic positioning. Second these users need to develop a strong relationship based on trust to their suppliers, as the project goals are not completely fixed in detail at the beginning of a project, but rather are developed during the project.

The role of implementation consultants needs to be extended: They must not only provide deep technical know-how about their enterprise systems, but they need to evolve to domain experts to provide proficient expertise in identifying and designing the business processes, which improve the prioritized business goals. The qualification of the customization team needs to be very broad, as it needs to know in depth the implications of the business requirements from a technical viewpoint.

Conclusions

Within a newly started research project the AI²M methodology will be further refined and evaluated in the field (SMILE Consortium, 2002). This will be done in cooperation with a vendor of a standard CRM system and a group of CRM consultants. Within the project a survey among SMEs, CRM system vendors and CRM consultants will be conducted in summer 2002 to increase the empirical basis regarding adoption of agile practices in the field and to gain insights into critical success factors for implementation projects at SMEs.

While the presented basic AI²M model is domain neutral, further refinement of AI²M will focus on CRM specific extension. These are foreseen for example for the business analysis phase, where a toolbox of CRM specific instruments (e.g. questionnaires, guidelines) shall help in rapidly identifying relevant improvement areas for an iteration cycle. Furthermore a tool supporting AI²M will be developed. In a second project phase, AI²M will be evaluated by implementing CRM systems at 10 SMEs. This shall provide an opportunity to collect experience data from the field.

Acknowledgements

This work was partly supported by the European Commission within the IST program under contract no. IST-2001-34376. Further information about the supported project SMILE – SME oriented methodology for the implementation of CRM systems with low effort – can be found at (SMILE 2002).

References

Adam, F., and O'Doherty, P. "Lessons from enterprise resource planning implementations in Ireland - towards smaller and shorter ERP projects," Journal of Information Technology, Vol. 15 , No. 4, 2000, pp. 305-316.

Agile Alliance "Manifesto for Agile Software Development.", http://agilemanifesto.org/, 2001, Accessed at 2002-03-07.

Beck, K. "eXtreme Programming Explained," Addison Wesley, Englewood Cliffs, N.J., 2000.
Boehm, B. "Spiral Development: Experience, Principles, and Refinements," Presentation at the Spiral Development Workshop at the Software Engineering Institute, http://www.sei.cmu.edu/cbs/spiral2000/february2000/BoehmSR.html, 2000b, accessed at 2002-05-12.
Boehm, B. "Software Management: Requirements that handle IKIWISI, COTS and Rapid Change," Computer Vol. 33, No. 7, 2000a, pp. 99-102.
Davenport, T.H. "Putting the Enterprise into the Enterprise System," Harvard Business Review, No. July/August, 1998.
Fichman, R.G. and Moses, S.A. "An Incremental Process for Software Implementation," Sloan Management Review, No. Winter, 1999, pp. 39-52.
Francalanci, C. "Predicting the implementation effort of ERP projects: Empirical evidence on SAP R/3," Journal of Information Technology Vol. 16, Nr. 1, 2001, pp. 33-48.
Grenning, J. "Launching Extreme Programming at a Process-Intensive Company," IEEE Software, Vol. 18, Nr. 6, 2001, pp. 27-33.
Kirchmer, M. "Business Process Oriented Implementation of Standard Software," 2nd edition, Springer, Berlin, 1999.
Parr, A. and Shanks, G. "A model of ERP project implementation," Journal of Information Technology, Vol. 15, 2000b, pp. 289-303.
Parr, A. and Shanks, G. "A Taxonomy of ERP Implementation Approaches," Proceedings of 33rd Hawaii International Conference on System Sciences (HICSS), IEEE, Maui, Hawaii, 2000a.
Paulk, M.C. "Extreme Programming from a CMM Perspective," IEEE Software Vol. 18, No. 6, 2001, pp. 19-26.
Poole, C. and Huisman, J.W. "Using Extreme Programming in a Maintenance Environment" IEEE Software Vol. 18, No. 6, 2001, pp. 42-50.
Schneider, G. and Winters, J.P. "Applying Use-Cases - A Practical Guide," Addison Wesley Longman, Reading, Massachusetts, 1998.
Schuh, P. "Recovery, Redemption, and Extreme Programming," IEEE Software, Vol. 18, No. 6, 2001, pp. 34-41.
SMILE Consortium "SMILE Project Homepage", http://www.vis.iao.fhg.de/smile, 2002, accessed at 2002-05-10.
Sprott, D. "Componentizing the Enterprise Application Packages" Communications of the ACM Vol. 43, No. 4, 2000, pp. 63-69.
Sumner, M. "Risk factors in enterprise-wide/ERP projects," Journal of Information Technology, Vol. 15, No. 4, 2000, pp. 317-327.
Szyperski, C. "Component Software - Beyond Object-Oriented Programming," Addison Wesley, Harlow, UK, 1998.