

Applying Agile Methods in Rapidly Changing Environments

Peter Kutschera

IBM Unternehmensberatung GmbH
Am Fichtenberg 1, D-71803 Herrenberg

Steffen Schäfer

IBM Unternehmensberatung GmbH
Leopoldstraße 175, D-80804 München

1 Introduction

Software development approaches have changed significantly throughout the last decade. Until the recent emerge of e-business, software development projects were mainly targeted at the implementation of well-known business processes. Projects took typically several months or even years to complete. After completion, only those projects were considered successful which implemented all of the given requirements correctly and completely. Due to the perception that it is economically much cheaper to detect errors in the project's life cycle early, a 'classical' software engineering approach for software development projects has been used. The idea was to flawlessly fix the given requirements and then to set up a relatively detailed design before starting with the implementation. This has been accomplished by writing comprehensive documentation which was later thoroughly reviewed to find as many errors as possible. Although this method has been applied somewhat successfully to software development projects in the past, the approach is not suitable for commercial projects in the era of e-business where things are moving fast. Due to the fact that many project development efforts are using leading-edge technology that is not yet well understood or even evolves during project lifetime and the underlying business models typically rely on a quick time-to-market, it is impossible to pin down a complete requirements list at the onset of a project. Project goals and system functionality need to be frequently adapted, in order to stay competitive in marketplace. Limited timeframe of many projects, of sometimes only a few weeks instead of months, as well as the necessity to quickly respond to changing business needs, are all demanding for a different approach for software development. This is why a lot of well-known methodologists proposed "lightweight" approaches for software development during the past two years or so. In February 2001, those methodologists formed the "Agile Alliance" and presented their manifesto [1] for software development. Due to this fact, it is expected that a larger community in software industries will regard the agile software development approach with its evolutionary aspects as a viable alternative to the classical software engineering approach for projects in changing environments.

The approach for agile software development presented in this paper is targeted at commercial software development projects and is based on IBM's Global Services Method. We use a more traditional, but incremental software engineering approach as a base and adapt this to specific needs in rapidly changing environments. We have used this approach because we believe that certain software development methodologies can be adapted to become agile. Also, our project teams are able to use a development method they are familiar with, rather than applying a totally new one. The adaptation consists of two steps. As a first step, the number of artifacts which have to be produced throughout the project is significantly reduced to put more focus on working software than on documentation. Second, we introduce a project organization with short iterative releases, in order to enhance the client's opportunity to provide feedback and to drill down or introduce additional requirements. Besides the use of a methodology which is able to cope with requirement changes, the underlying software architecture plays an important role in the overall development project, because it must be flexible enough to incorporate new requirements without breaking the code that has already been developed. The issue of designing a flexible software architecture is addressed in this paper, before we conclude with a case study where the usage of the approach presented in this paper has been successfully demonstrated.

2 Related Work

All of the methodologies for agile software development that have been introduced during the last years can either be categorized as meta process methodologies or specific development methods. *Meta process methodologies* do not describe specific development approaches, but rather focus on general procedures to support the development team in establishing a project-specific development approach. Adaptive Software Development [6], the family of Crystal methodologies [5] and Scrum [2] belong to that category. In contrast to the meta process methodologies, specific development methods like the Dynamic Systems Development Methods (DSDM) [8], eXtreme Programming (XP) [7] or Feature Driven Development (FDD) [4] describe proven practices or give concrete hints to guide a development team to implement software that is adaptable to changes.

Although a lot of different agile methodologies have been proposed in the past, almost no references about real-world software project using agile methods can be found in the literature. One exception to the rule is the famous Chrysler C3 payroll system [3] where XP has been used for the first time and successfully been applied. Thus, one of the goals of this paper is to fill this gap.

3 The Principles of Agile Software Development

In this section, the principles of agile software development are introduced. So, the goal of adopting a standard software development method which is described in greater detail in the next section, is to achieve these principles.

Besides the fact, that these principles illustrate the essence of agile software development, they present the prerequisites for our tailoring approach.

The following principles of software development are taken from the Agile Manifesto [1]:

- *Individuals and interactions over processes and tools.* This means, the successful outcome of a project depends more on the interaction of skilled professionals than on the usage of a well-defined process or the latest tools. Although this point is valid, we regard this issue beyond the scope of this paper.
- *Working software over comprehensive documentation.* This statement addresses the need to reduce comprehensive documentation, because an extensive documentation does not mean that the actual problems have been well understood. In addition, it incorporates significant overhead every time requirements are added or have to be changed.
- *Customer collaboration over contract negotiation.* The key message of this statement is that collaboration with the client is one of the critical success factors of a software project, because through active collaboration the client can help the team to understand the wants and needs.
- *Responding to change over following a plan.* Generally speaking, making a plan and following it, is not a bad practice. This statement covers a slightly different aspect where changing requirements are not taken into account, because they do not fit according to the project plan. Obviously, to deliver a system in time that implements requirements no longer important to the user, is useless. The solution to this problem is the usage of short release cycles, together with the client's ability to introduce new requirements or change priorities.

Based on the principles for agile software development, the conclusion is to use only as much documentation as really necessary, to introduce short release cycles and to involve the customer as much as possible, as a reviewer and domain expert throughout a software development project.

4 Applying Agile Principles to a Standard Software Development Method

4.1 General Approach For Software Development

In this section, we first outline IBM's Global Services Method which we use for software development projects, before the method adoption for rapidly changing environments is presented in greater detail.

Basically, our software development method has two major elements: a set of work products and a recommended, but adaptable work breakdown structure. *Work Products* are artifacts that are produced during a project and can be final deliverables of a project as well as internal project-specific results. Every Work Product has a well-defined purpose and is produced by team member with specific roles (e.g. business analyst, IT architect, etc.). The *Work Breakdown Structure* hierarchically divides the project into phases, activities and tasks which are performed by team members acting in the various roles.

A typical custom application development project consist of the five phases depicted in Figure 1:

1. During *solution outline phase* the project scope and approach are defined. Requirements are gathered on a high level, and an initial architecture, as well as a project plan are developed.
2. The purpose of the *macro design phase* is to drill down on the initial set of requirements gathered before, to refine a functional and operational architecture and to perform the installation of the development environment.
3. The *micro design phase* is used to refine the existing macro design which means that requirements are analyzed in more detail for a specific release and the architecture / design are further developed.
4. During the *build phase*, the design is refined, the source code is crafted, documented and tested. In addition, any educational material to train end users is produced.
5. In the *deployment phase*, the acceptance tests are run, the transition to the production environment is performed and the next iteration is planned.

On a typical project Micro Design, Build, and Deployment are preformed multiple times, ie. once per release. To be absolutely precise, Build Cycles are even conducted multiple times within a release, building short term increments. In a nutshell, we start off with a truly iterative and incremental software development method, comparable to the Rational Unified Process (RUP).

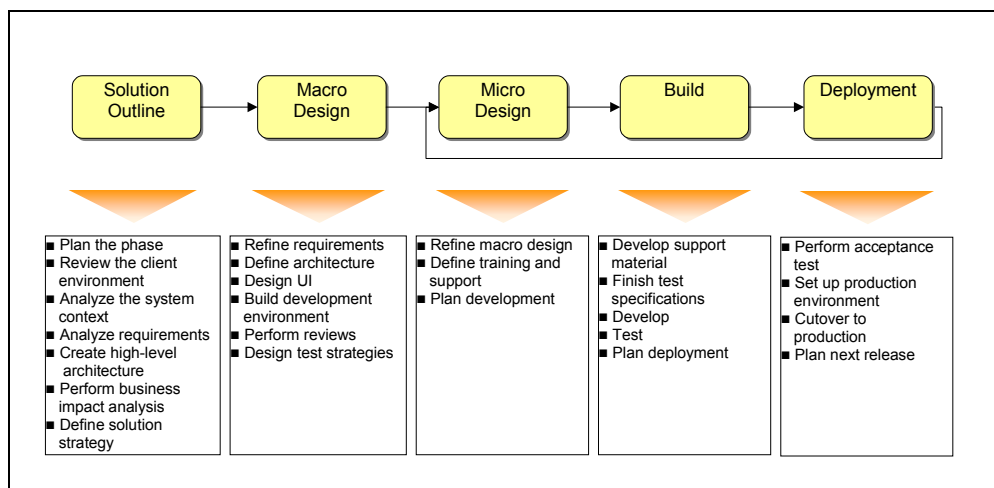


Figure 1: Standard Approach for Software Development Projects

4.2 Method Adoption: Which Artifacts should be used?

The first step of the method adoption for an agile process consists of the selection of artifacts that should be used in the project. IBM Global Services' method consists of more than 150 artifacts, and on a large, 'traditionally' run project most of them are produced.

We believe the following documentation is essential for every serious real-world agile software development project:

Purpose	Name of the artifact	Description
Requirements	Non-functional Requirements	The non-functional requirements incorporate a list of aspects like scalability, response time, etc.
	Use Case Model	The functionality of a system is described by sequences of use cases. This description might be rather short, similar to the user stories that are mentioned in the XP methodology, including a sketch of the user interface and the functionality behind the different user actions (referencing other use cases). Additional refinement is possible during the iteration where the use case gets implemented.
System Architecture	System Context Diagram	This diagram gives a high-level overview of the different components of a system (e.g. browser, application server, ERP package, etc.) as well as the different type of users. It is helpful to the client to understand the architecture of the system.
	Architecture Overview Diagram	In contrast to the System Context Diagram which treats the system under construction as a black box, the Architecture Overview Diagram depicts a high-level view inside the system and thus shows the major logical building blocks.
	Component Model	This artifact gives a more detailed description of the software components. It describes each components' responsibilities and interfaces. The level of detail applied here, depends on the experience of the project team. In a small team with highly experienced developers, this description can be rather short.
	Operational Model	The Operational Model is a diagram which depicts the hardware infrastructure that is used to meet the software's non-functional requirements. In addition, it shows the mapping of software components to the underlying hardware.
	Standards	A simple list of standards that are mandatory in the context of a project (e.g. LDAP for authentication, the Java programming language).
Software	Release Plan	A release plan which is a classical project management activity incorporates aspects like the duration of a release or certain tasks, the number of people needed to complete a release, which skills must be available during a release.
Standards	Coding Guidelines	Coding guidelines are mandatory for all of the software developers.
Release Planning (part of every iteration)	Increment Goals	A list of the features which are part of every iteration.
	Deployment Plan	This artifact could be called an installation guide which describes how to deploy the working software to the production system.
Test Planning	Test Strategy	The test strategy described the purpose, the time and the people

		that involved in the testing activities.
--	--	--

We believe that the list of artifacts above is minimal, but to further stress this point, we'd like to clarify: The *requirements documentation* can be rather short. Every use case can be described on a single page or even a list of bullet points. All of the non-functional requirements could be typically addressed in a maximum of five pages.

- The *system architecture* should be a maximum of 15-20 pages, otherwise nobody will ever read it. The documentation should be easy to comprehend and accessible to all team members.
- The *standards* consist of coding guidelines which are relevant for the developers only.
- *Release planning* is a typical project management activity which must be done.
- A *test plan* describing the general test strategy can be rather brief but must be documented. A few pages will be enough.

Just by using good development environment it is possible to abandon a lot of documentation. Extensive design documentation, like class - or sequence diagrams in object-oriented projects, are no longer necessary to be kept up to date manually when good tools can be used. Instead, class diagrams or sequence diagrams are sketched on a piece of paper, implemented, and then thrown away. Leading-edge CASE-Tools let developers reverse engineer their code and depict what is coded in more accessible UML diagrams. Testing tools can support automated regression tests and thus reduce the amount of detailed test specifications up front. JUnit is frequently used for unit testing and is extremely efficient. Integrated Development Environments (IDEs) with code browsing facilities (e.g. WebSphere Studio Application Developer) make code easily accessible and further reduce the amount of required documentation. It is important to note however, that the source code itself should be very well documented! Source code reflects the ultimate design, and hence, comments there should be extensive.

As a conclusion, it can be stated that the appropriate selection of artifacts helps to reduce the documentation to an acceptable level and to focus more on working code. Key documents describing requirements, or the high-level architecture of the system should be easy to understand and must be kept up to date.

4.3 Method Adoption: What is the appropriate Project Organization?

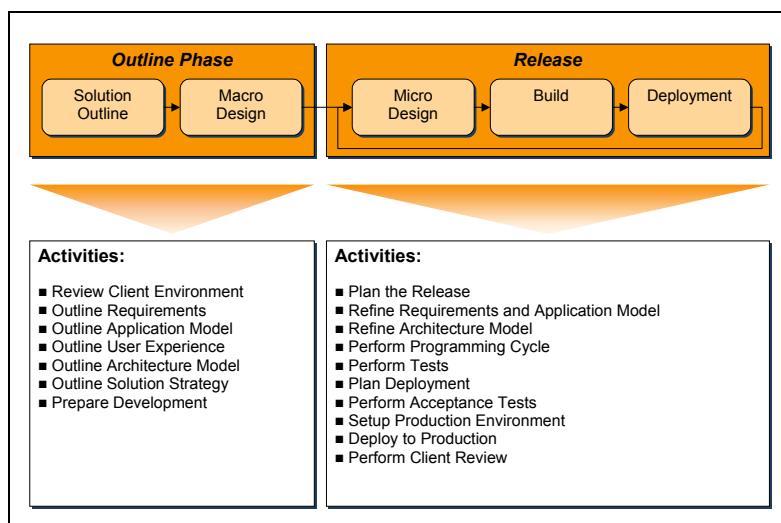


Figure 2: Proposed Project Organization for Agile Software Development

The second step of the method adoption consists of finding an appropriate project organization for agile software development. A proposal is presented in Figure 2 which consists of two phases, in contrast to the five phases of the IBM Global Services' method. Only crucial activities and tasks are distilled into those two phases, in order provide a really light weight approach. So, a project starts by performing an initial outline phase, combining the solution outline and macro design phase, followed by short iterative release cycles, combining the micro design, build and deployment activities.

In general, the initial *outline phase* might not take more than two months, this should be enough to define the scope of the project, gather the most important requirements and design a high-level architecture of the system. Thus, the outline phase consists of the following activities:

- *Review client environment*: The goal of this activity is to obtain or document the clients' IT standards and to identify any pre-determined components of the future IT architecture.
- *Outline requirements*: During this activity the system context is established to describe the main objects that make up the environment and the system being developed, it identifies what is part of the system and what is not. This activity is also used to gather the functional requirements as use cases. The non-functional requirements which are gathered during this activity, too, are used to understand the complexity and performance requirements of the system and are key drivers of architecture and infrastructure design.
- *Outline application model*: The application model consists of the business objects and their interactions. Because the goal of this activity is to understand the scope and complexity of the application, this activity results in a high-level object model capturing the responsibilities of the major business objects as well as their interactions.
- *Outline user experience*: This activity is used to create a high-level understanding of the interactive aspects of the application. During this activity the user interface design guidelines as well as the user interface architecture are to be defined.
- *Outline architecture model*: The goal of this activity is to formulate an initial vision of the overall system which makes it possible to evaluate alternative high-level architectural overviews and choose between them. During this activity, it makes sense to identify relevant assets and check the possibility to re-use them in the current project context. The functional aspects of the architecture is captured in the high-level component model whereas the high-level operational model is used to describe the operational aspects. Although only high-level architectural models are created, it is important that these models are designed to fulfill the clients' performance requirements. Finally, this activity is used to decide which commercial products should be acquired to support the development of the application.
- *Outline Solution Strategy*: This activity is meant to ensure a common understanding of the project as well as to provide a basis for performing the estimations to able to plan the development effort. During this activity the test strategy gets defined and the release and the deployment plans are outlined.
- *Prepare development*: Before starting with development of the software increments, obviously the actual development environment must be set up. This means that the necessary hardware and software has to be acquired and to be installed. Additionally, the coding guidelines for developing source code must be defined.

After completion of the outline phase, iterative development of a release can begin. A *release cycle* can be completed in, say six to ten weeks time including release planning, refinement of the requirements, implementation and deployment of the software. In detail, the following activities are part of every release cycle:

- *Plan the release*: As the first step of this activity, all of the remaining requirements have to be prioritized by the client according to their importance. The goal of this step is to ensure that the most important requirements are addressed as part of the next release cycle. Then, the duration to complete the most important requirements is estimated to define the scope of the next release cycle. The outcome of this activity is a release plan consisting of all the requirements addressed by the current release and the time of their completion.

- *Refine requirements and application model*: The purpose of this activity is to detail the requirements for the upcoming release. So, the system context, the use cases and the non-functional requirements are finalized as part of this activity.
- *Refine architecture model*: During this activity the component and operational model for the current release are completed to provide the basis for a detailed design and implementation.
- *Perform programming cycle*: This activity consists of tasks to design the object model, build and test the source code. This is the actual coding.
- *Perform tests*: The goal of this activity is to test the functionality of a component which consists of internal logic and design, exception handling and code coverage. Basically, developer tests are accomplished by performing source code reviews and conducting unit tests. To test the proper interaction between the various components, intermediate releases are built so that integration level tests can be performed.
- *Plan deployment*: As part of this activity, the plan to guide and control the roll-out of the system is refined and the code is physically packaged for installation.
- *Perform acceptance tests*: The purpose of this activity is to execute the application in a production-like environment and to verify the system meets the clients' functional and technical requirements. Thus, as part of this activity user acceptance tests and system integration tests are conducted.
- *Setup production environment*: During this activity the necessary software and hardware infrastructure are set up to install the components of the application. Additionally, all of the data sources which are used by the application have to be installed.
- *Deploy to Production*: After setting up the production environment, the application is finally brought into the production environment and again thoroughly tested.
- *Perform Client Review*: The goal of this activity is to receive and analyze the client feedback which has been given during the integration and acceptance tests. All of the requirements that must be re-worked should have top priority in the next release cycle. This activity concludes the release cycle and lays the groundwork for the subsequent release.

As a conclusion it can be stated that by using short iterative release cycles and involving the client frequently in the development project by the means of reviews and feedback activities, it is possible to allow for requirement changes during a project's life cycle.

5 Designing a flexible Software Architecture

Based on our experiences in different software development projects, many projects fail due to the fact that they are trying to establish a detailed architecture long before software development actually starts. However, for applying an agile software development method, an iterative approach for designing the software architecture is also a must. This can be accomplished by defining a high-level software architecture which captures all of the following strategic aspects during the outline phase that has been introduced in the last section:

- *Durability*: Using state of the art technology and considering open standards ensure the viability of the architecture in the future.
- *Stability*: Utilizing components as the underlying design concept for the software makes it possible to change the implementation of the components without breaking their contracts.
- *Flexibility*: Taking different alternatives for the realization of a software component into account (a custom application development versus package integration approach) helps minimizing future risks.

This strategic architectural view presents the framework for the iterations during the release phase. Now, during every iteration tactical decisions regarding the realization of the different components are made. A tactical decision could be, for example, to develop the first version of a component during an iteration and change the custom code in a later iteration of the project when a commercial solution becomes available. Thus, on one hand,

tactical decision are used to be able to deliver software in a timely fashion which corresponds with the overall strategic architectural view. On the other hand they are used to refine and validate the strategic architectural view which becomes more and more consolidated over time.

6 Case Study

As one brief example, we present a software development project that built a mobile device portal for a Finnish Wireless Service Provider. The project got started in May 2000 and was mission critical, with an ambitious schedule and a fixed deadline. Development effort has been split over various subprojects, with a 25+ developers team developing the portal platform, and some twenty smaller teams implementing the actual applications. Due to legal reasons related to operator licensing, the portal had to be online by April 2001. Original project management chose an ad-hoc software development approach, neglected requirements gathering, unit testing, and other good practice, so the project came in trouble. Project management and development approach was then changed in January 2001. We then started to apply the approach proposed in this article with great success.

The first release of the mobile portal got deployed beginning of April 2001. After this initial launch, additional releases, improving on the platform and consisting of 5-8 new applications got launched every month. A range of XP programming practices has been successfully applied: solving problems in the simplest way, constant refactoring of code, pair programming, rigid unit testing using the JUnit testing tool, emphasis on well documented source code over producing separate documentation and complete builds at least once a day.

Thus, by applying agile software through adoption of an iterative development approach which the team was familiar with and the principle to compromise scope over delivery date, the project was successfully completed.

7 Conclusion

In this article an approach for tailoring a standard iterative development methodology has been presented. The key issues for using such an approach for agile software development were the reduction of the number of artifacts to an acceptable level and a proposal for an appropriate project organization which improves customer collaboration and is adaptable to changing requirements. The feasibility of our approach has been demonstrated by applying it in a real-world mobile portal project in the area of telecommunication.

8 References

- [1] Agile Alliance: *Manifesto for Agile Software Development*, available at <http://www.agilealliance.org>
- [2] M. Beedle, K. Schwaber: *Agile Software Development with SCRUM*, Prentice Hall, 2001
- [3] The C3 team: *Chrysler goes to the Extremes*, Distributed Computing, pp.24-28, October 1998. Also available at <http://www.xprogramming.com/publications/dc9810cs.pdf>
- [4] P. Coad, E. Lefebvre, J. De Luca: *Java Modeling In Color With UML: Enterprise Components and Process*, Prentice Hall, 1999
- [5] A. Cockburn: *Agile Software Development*, Addison Wesley, 2001
- [6] J. Highsmith: *Agile Software Development Ecosystems: Problems, Practices, and Principles*, Addison Wesley, not published yet
- [7] R. E. Jeffries, et al: *Extreme Programming Installed*, Addison Wesley, 2000
- [8] J. Stapleton: *DSDM – Dynamic Systems Development Method*, Addison Wesley, 1997