The Right Process For Each Context: Objective Evidence Needed

Ove Armbrust
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
+49 631 6800 2259
ove.armbrust@iese.fraunhofer.de

Dieter Rombach
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
+49 631 6800 1001
dieter.rombach@iese.fraunhofer.de

ABSTRACT
The growing importance of software in ever more technical systems has led to new demands with respect to developing software. The demand for more functionality, higher quality, and faster delivery hence poses major challenges to the software industry. The software process community has responded with a variety of different development processes such as the waterfall model or the incremental commitment model, however, the number of late or failed projects has not decreased as much as it was desired. In the new millennium, agile development approaches promised a new way out of this dilemma. After several years of heated discussions, it is now time to evaluate applicability, advantages, and challenges of different software development approaches based on sound, empirical evidence instead of anecdotes and hearsay. This paper briefly investigates the major differences between agile and traditional approaches, illustrates the difficulties in selecting the "right" approach for a given project, and proposes hypotheses for empirical evaluation, in order to build a solid body of knowledge that can be used for said selection.

Categories and Subject Descriptors
D.2.9 [**Management**]: Software process models (e.g., CMM, ISO, PSP)

General Terms
Management, Documentation, Human Factors, Measurement, Experimentation

Keywords
Software process models, agile, rich

1. INTRODUCTION
The importance of software and its influence on key features of an enormous variety of products has risen steadily for the past decades. Hence, more software had to be developed, for more application domains, providing more functionality, and replacing more mechanical and hardware parts of systems. This increased the variety of software quality demands and raised the bar regarding when such a quality demand is deemed fulfilled.

ICSSP'11, May 21–22, 2011, Waikiki, Honolulu, HI, USA.
ACM 978-1-4503-0580-8/11/05.

The software process community has come up with a great variety of different approaches and technologies to create software that satisfies these demands. Approaches like the waterfall approach [1], the spiral model [2], the incremental commitment model [3], or the Cleanroom approach [4] are well-known and taught in any software engineering class around the globe.

However, even though the software process community has been very hard-working to provide all these approaches, there still were a large number of software projects that did not achieve their goals. The often-quoted CHAOS report by the Standish Group [5] indicates that while the number of software projects completed on time has about doubled from 1994 to 2008, still only about one third of all projects fall into this category – all other projects were either late and over budget, or cancelled altogether (Figure 1).

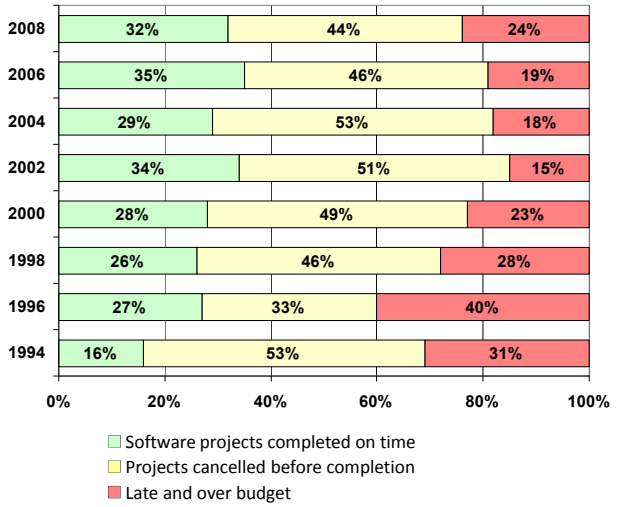

Figure 1: CHAOS Report overview

With the emergence of lightweight, agile, highly iterative approaches for software development, a new viewpoint was added to the discussion. Maybe ever more elaborate, precise, and comprehensive development approaches are not the answer to the challenges, but the opposite? Maybe a development approach that is reduced to the absolute bare necessities could deliver the required quality without exploding costs and effort? Approaches that emerged under the label of "Agile Development" promoted this idea, for example, the Extreme Programming approach [6] or Scrum [7].

Quickly, the software process community was debating intensively whether or not agile approaches were the answer we had been looking for, instead of the classic, heavyweight processes that were the method of choice so far. Occasionally, this debate almost resembled religious disputes, with heated arguments pro and against each approach. What made the discussion even more difficult was the fact that "agile" did not describe one single approach, but a whole class of different approaches, each of them different in many ways. What is common to all agile approaches, though, are two major features:

1. *Highly iterative scheduling.* Typical agile development projects have release cycles from anywhere between two and 8 weeks. This is radically different from classic, phased projects, where one phase can easily take many months, and the first release of the product may not happen for years.
2. *Very little documentation.* Agile projects try to avoid as much documentation as possible. This also radically differs from classic approaches, which traditionally put great emphasis on detailed documentation of requirements, design etc. – naturally, agile approaches were immediately popular amongst developers, because they cut away most of the annoying part of developing software. Ideally, a software product that was produced in an agile way is documented only through its code and tests.

Now that agile approaches have existed for a decade, the number and fierceness of heated debates has decreased significantly. Agile approaches have established themselves as a viable alternative to traditional approaches. However, what is still missing in many cases is reliable information on strengths and weaknesses of each approach, and clear guidance when to use which approach. We believe that this discussion should be based on facts, not on anecdotal stories or hearsay.

2. OBJECTIVE EVIDENCE NEEDED

What are the major reasons for the many of the discussions around process models, technologies, and tools in the software engineering community? It's the inability of the software industry to deliver products with the desired quality without letting effort and/or cost getting out of hand. Optimizing each parameter individually is not the problem – but their combination seems to be more difficult than anticipated.

So what is the problem? The problem is that during the development of software, defects are introduced that let the actual product deviate from its envisioned ideal, i.e., the product that the customer would like to receive. This is a natural phenomenon and accepted by most organizations: People make mistakes, and some of these mistakes introduce defects into the software they are working on. However, since the customer typically does not like defects in his software, they must be removed again. From a purely technical point of view, it does not matter when a defect is introduced, and when it is removed. However, if cost and effort play a role, it does: the earlier a defect is found and removed, the fewer additional defects it may trigger, and the cheaper its removal usually is.

To illustrate the different approaches' way of resolving this issue, let us review two examples. The first example represents the traditional approaches typically described in "rich" process models, the second example considers an "agile" process model.

For the first example, let us consider a waterfall development project split into four major phases (Figure 2):

1. Requirements elicitation
2. Design
3. Coding
4. Testing

At the end of each phase, defect detection activities are performed. If defects are found, they are removed, i.e., all or a selection of the phase's activities are repeated to remove the defects (solid backlinks in Figure 2). If no more defects are detected, the project moves on to the next phase. However, it may also become necessary to go back even further, for example from the end of the testing phase to the software design (dashed backlink). This happens if a design defect was not found during the design or coding phase.

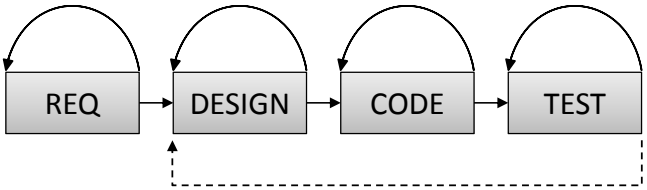

Figure 2: Traditional development approach

The separation of the project into logically disjoined phases thus is a means of preventing the spread of defects throughout development, and eventually of reducing defect detection and removal cost. This kind of "divide and conquer" strategy is very well known and applied in software engineering: in this case, the problem is cut vertically into smaller, manageable units.

For the second example, let us consider a similar development project, but using an agile approach. Agile approaches also divide the problem (i.e., the software system to be developed) into smaller, manageable units. This means that they also use a "divide and conquer" strategy, but it differs significantly from the waterfall approach described previously.

An agile approach cuts the problem horizontally, meaning it does not cut between logical development steps like a traditional approach, but based on functionality to be delivered. Each phase (often also called "sprint) within an agile project delivers a subset of the final product's functionality, and as part of that phase performs all the software development activities that are necessary. In our example, this would be requirements elicitation, design, coding, and testing. One guiding principle is to only do what is required to reach the current phase's goals. This means, for example, that a design is chosen that just satisfies the current requirements – and ideally nothing beyond.

If a defect is found within such a phase, it is immediately corrected, updating all necessary parts of the software product. The solid arrows in Figure 3 indicate this case. However, all decisions made within a specific phase influence all later phases – yet, they are typically made with only the requirements of the respective phase in mind. The dotted arrow in Figure 3 indicates this case. This typically leads to frequent design changes

throughout development, every time requiring changing all dependent entities like code and test cases.

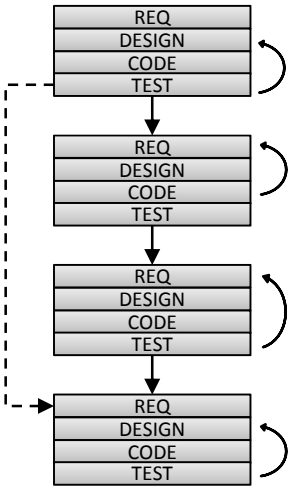

Figure 3: Agile development approach

So how do the traditional and the agile approach compare? With respect to divide and conquer, they actually pursue the same strategy: cut the problem into smaller, manageable pieces, in order to increase the effectiveness of defect detection and removal, and in order to limit the effort required for these activities. If the individual pieces of the problem are still too big, i.e., too many defects escape into the next phase, they also pursue the same strategy: Make the pieces smaller. For example, traditional approaches then divide the design phase into preliminary and detailed design phases, agile approaches reduce the functionality that is implemented in one phase.

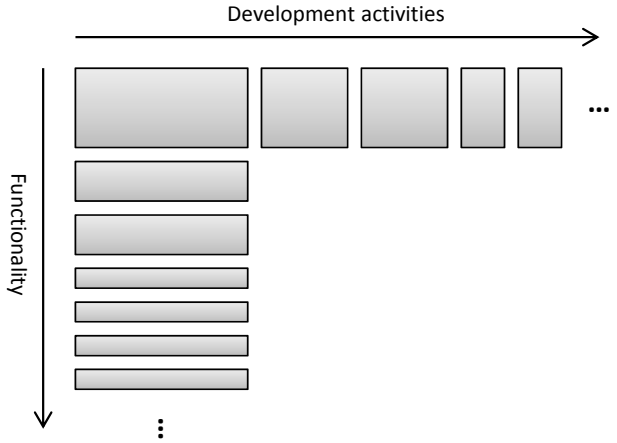

Figure 4: Traditional and agile divide and conquer strategy

Figure 4 compares the traditional and the agile divide and conquer approach. When too many defects escape into the next phase (i.e., the box is too big), the problem that is addressed within a single phase is decreased in size (i.e., the box is made smaller). Whether to cut it vertically or horizontally is less important than to cut it at all.

If the first major difference between traditional and agile development approaches is not so major after all, maybe the second is? So let us take a look at the product documentation that is produced during development. The traditional approaches typically produce a very comprehensive documentation: detailed requirement specifications, design, and test documentation for example. The agile approaches are clearly different in this respect: requirements are mostly documented on a very abstract level in terms of user stories, there is typically no separate design documentation, and the test cases basically specify what the software is supposed to do.

However, this self-limitation with respect to documentation is more or less a conscious decision, and not a compulsory feature of an agile approach. Put in other words: If we take the iterative enhancement approach described by Basili and Turner in 1975 [8] and reduce the amount of documentation demanded, we get very close to what is nowadays known as "agile development". Hence, we believe that there are not two groups of development approaches (agile and heavyweight), but rather a great variety of different shades between the two extremes.

One major question remains, however:

3. WHEN TO USE WHICH PROCESS?

This really is the question that made the software development community discuss (and sometimes fight) during the past decade. What drove the discussions was the fact that it is rather easy to find many good arguments for or against any development process.

Example 1. Let us assume that a project has unclear requirements, because the customer has some vague idea of the system he would like to receive in the end, but is not entirely sure. However, he wants to use the system as quickly as possible, because it will give him a significant business advantage – others are trying to build a business on the same idea, and whoever enters the market first has a head start. This context indicates that a highly iterative, fast development approach is beneficial. The typical agile process with close customer involvement, frequent delivery and short feedback cycles seems much better suited than a traditional heavyweight approach that first elicits all requirements, then does the design, coding, etc. – by the time the traditional approach gets the product on the market, the iterative approach has produced at least three versions already. So, a clear case for an agile process.

Example 2. Let us assume a second project in a regulated context, for example, aerospace. In order to be able to sell the software, it must be certified according to a strict standard. One requirement of this standard is that the parts of the design covering functionality related to the well-being of humans (e.g., pressure control) must be formally verified, i.e., a formal proof must be supplied that the design is correct. With an agile approach, this proof would have to be provided after every design change, or one would run the risk that when the product is finished, the proof fails, which would mean that the product cannot be introduced into the market. Hence, this context clearly favors a traditional, possibly waterfall-style approach, in order to guarantee certification with reasonable effort.

Examples such as the two described can be provided for pretty much any development process variant, from extremely agile to extremely formal. We thus firmly believe that the software process community should not focus on confirming that some approach is "better" than the other, but on objective evidence regarding the different approaches' suitability *for specific*

contexts. In fact, we believe it is of paramount importance to systematically collect, analyze, and use empirical evidence with respect to the applicability, benefits, and drawbacks of different development approaches. From our point of view, this is the only way to steer clear of dreadful "I believe"-style discussion, and to implement sound engineering practices with respect to software processes.

4. HYPOTHESES

Many of the discussions with respect to the right choice of development process are based mostly on anecdotal information. Some of these anecdotes are (in no particular order, and not claiming to be exhaustive):

- Agile approaches deliver more elegant designs than traditional approaches.
- Agile approaches make people more productive compared to traditional approaches.
- Agile approaches deliver better quality software than traditional approaches.
- …

For every anecdote, there is usually some kind of study/report that seems to support the respective claim. However, what is often missing is conclusive, empirical evidence under which circumstances, and in combination with which other influencing factors the reported experience was made. This leads to widely varying reports. For example, [9] reports that introducing XP into a large organization was rather difficult, whereas [10] reports the opposite.

When looking at productivity, a number of studies report that agile approaches are more productive than traditional approaches [11] [12] [13], however, [14] reports that traditional approaches show higher productivity than agile methods. Similar differences can be found, for example, for employee satisfaction, customer satisfaction, the ability to incorporate (late) changes in the product, the interchangeability of team members, produced code size and quality, the uniformity of work procedures, and other factors. [15] gives a comprehensive overview.

Because of these contradictory reports, we propose to define a number of hypotheses that should subsequently be tested with systematic, empirical studies. Apart from defining the "right" hypotheses (i.e., those that provide information that is of high value for research and industry), it is also important to precisely state the subject of interest. As stated before, "agile" describes a wide variety of approaches – considering the process community's limited resources, is seems prudent to focus on selected, clearly defined and separated agile approaches.

In our opinion, obvious candidates are Scrum and XP. Scrum is one of the few lightweight and agile, yet very precisely defined processes; its application has continuously risen over the past years. XP is very popular and widely used across all software developing organizations and should thus not be neglected, either. However, since XP mostly consists of rather generic principles and a sketch of an iterative development lifecycle, real-life implementations of XP also differ widely. This should not be neglected, because the principles cannot be viewed independently, but influence and rely on each other [16]. Hence, the degree to which XP is implemented must be carefully recorded, in order to allow for analyses and comparisons.

As a starting point to build a solid body of (trusted) knowledge, we propose a number of hypotheses which should be examined through empirical studies. Please note that some of these hypotheses are directed, whereas others are not. This is due to the fact that for some of the circumstances reflected in the hypotheses, anecdotal reports seem to point into a specific direction, whereas for others, this does not seem to be the case.

We propose the following hypotheses for empirical studies:

H1. Agile approaches yield higher employee satisfaction than traditional approaches.

H2. Agile approaches yield higher customer satisfaction than traditional approaches.

H3. Agile approaches are more likely to push customers to the limit than traditional approaches. *(Note: There are reports that customers could not keep up the high level of involvement into agile projects over a longer period of time.)*

H4. Agile approaches can integrate late requirements changes better than traditional approaches.

H5. Members of agile teams are less interchangeable than members of traditional teams.

H6. Software architectures/designs created in agile projects are less consistent than when created in traditional projects.

H7. The work procedures of agile projects are more uniform than those of traditional projects.

H8. Software architectures/designs created in agile projects differ in quality from those created in traditional projects.

H9. Productivity in agile projects differs from that in traditional projects.

H10. The quality of software developed in agile projects differs from the quality of software created in traditional projects. *(Note: It should be carefully evaluated which quality aspects are considered.)*

H11. The size of the code created in agile projects differs from that created in traditional projects.

H12. The cost for developing a piece of software in an agile project is different from that in a traditional project.

5. LEAN VS. RICH PROCESS MODELS

We believe that the documentation of development processes ("lean" vs. "rich") is of secondary concern in this context. Again, there are good reasons to document the required activities in more or less detail. Certain safety certifications (for example, SILs as described in IEC 61508 [17]) require the application of very specific methods in order get the desired certification. In this case, it would be advisable to document these methods (and possibly the tools to be used, too) in great detail, in order to provide a clear guideline what to do.

In other cases, it may be advisable to reduce the amount of documentation detail. For example, if an organization cooperates with different customers and directly interfaces with their requirements management systems, a process documentation should rather lay out general guidelines on how requirements should be recorded (e.g., that they must be uniquely identifiable) than precisely prescribe templates or tools, which would lead to enormous conversion and synchronization effort.

Hence, we believe that both lean and rich process documentation have their value, depending on the respective context. The extent of process documentation should thus also correspond to the context.

6. PROPOSAL FOR PRINCIPLES AND VALUES

In line with our strong belief that what is needed in the first place is strong, empirical evidence of the applicability, advantages, and challenges of different software development processes, we propose the following values.

6.1 Proposed Values

- *Facts and evidence over belief and intuition.* It is of paramount importance to provide reliable evidence in order to be able to make sound, fact-based decisions.
- *The best-suited process for the respective context over silver-bullet-illusions.* Like in all other engineering disciplines, there is no one-solution-fits-all answer to the question of which process to choose. The earlier we acknowledge this fact, the earlier we can focus our efforts on determining relationships between contexts and processes.
- *Continuous improvement of processes over static standards.* While standards provide a very valuable common foundation and vocabulary, we believe that organizations' processes should not be static, rarely changed encyclopedia, but actively monitored, updated, and improved tools to achieve the organization's goals.

6.2 Proposed Principles

- We determine the context before we discuss processes.
- We base our decision for a specific process on objective evidence.
- We make our decision for a specific process transparent regarding rationales.
- We value feedback regarding the chosen process.

7. ACKNOWLEDGMENTS

This work was in part supported by the German Federal Ministry of Education and Research (BMBF), grant number 01IS09049B.

8. REFERENCES

[1] Royce, W. W. 1970. Managing the Development of Large Software Systems. In *Proceedings of IEEE WESCON, Los Angeles, CA, USA*, 1-9.

[2] Boehm, B. W. 1986. A Spiral Model of Software Development and Enhancement. *ACM Sigsoft Software Engineering Notes* 11, 4, 22-42.

[3] Boehm, B. W. 2009. The Incremental Commitment Model (ICM), with Ground Systems Applications. In *Ground Systems Architectures Workshop (GSAW), March 23-26, 2009, Torrance, California, USA*.

[4] Mills, H. D., Dyer, M., and Linger, R. C. 1987. Cleanroom Software Engineering. *IEEE Software* 4, 5, 19-25.

[5] The Standish Group: *Welcome to the Standish Group International*. URL: http://www.standishgroup.com/, last visited 2011-03-11.

[6] Beck, K., and Andres, C. 2005. *Extreme Programming Explained*, Addison Wesley, Boston.

[7] Schwaber, K., and Beedle, M. 2002. *Agile Software Development with Scrum*, Prentice Hall, New Jersey.

[8] Basili, V. R., and Turner, A. J. 1975. Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering* SE-1, 4, 390-396.

[9] Svnesson, H., and Höst, M. 2005. Introducing an agile process in a software maintenance and evolution organization. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05), Manchester, UK, March 21-23, 2005*.

[10] Bahli, B., and Zeid, E. 2005. The role of knowledge creation in adopting extreme programming model: an empirical study. In *Proceedings of the ITI 3rd International Conference on Information and Communications Technology: Enabling Technologies for the New Knowledge Society, April 17-19, 2009, Doha, Qatar*.

[11] Dalcher, D., Benediktsson, O., and Thorbergsson, H. 2005. Development life cycle management: a multiproject experiment. In *Proceedings of the 12 International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05), April 4-7, 2005, Greenbelt, MD, USA*.

[12] Ilieva, S., Ivanov, P., and Stefanova, E. 2004. Analyses of an agile methodology implementation. In *Proceedings of the 30th Euromicro Conference, Rennes, France, August 31-September 3, 2004*.

[13] Layman, L., Williams, L., and Cunningham, L. 2004. Exploring extreme programming in context: an industrial case study. In *Proceedings of the Agile Development Conference, Salt Lake City, UT, USA, June 22-26, 2004*.

[14] Wellington, C. B. T., and Girard, C. 2005. Comparison of student experiences with plan-driven and agile methodologies. In *Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference, Indianapolis, Indiana, October 19-22, 2005*.

[15] Dyba, T., and Dingsøyr, T. 2008. Empirical studies of agile software development: A systematic review. *Information and Software Technology* 50, 833-859.

[16] MacCormack, A., Kemerer, C. F., Cusumano, M., and Crandall, B. 2003. Trade-offs between productivity and quality in selecting software development practices. *IEEE Software* 20, 5, 78-85.

[17] International Electrotechnical Commission 2005. *IEC 61508, 'Functional safety of electrical/electronic/ programmable electronic safety-related systems'*, IEC, Geneva, Switzerland.