Constructing Agile Software Processes

Hasko Heinecke
Daedalos Consulting AG
Seestr. 510
CH-8038 Zürich
Switzerland
+41 (1) 481 07 20
hasko.heinecke@daedalos.com

Christian Noack
Daedalos Consulting GmbH
Ruhrtal 5
D-58456 Witten
Germany
+49 (2302) 9790
christian.noack@daedalos.com

Daniel Schweizer
Daedalos Consulting AG
Seestr. 510
CH-8038 Zürich
Switzerland
+41 (1) 481 07 20
daniel.schweizer@daedalos.com

ABSTRACT
A Software Development Process usually cannot be successfully applied out of the box. Practitioners know that methods have to be adapted to company culture and project constraints. However, many current Agile Processes lack an explicit description of the constraints, alternatives, and side effects of such changes. This paper describes how a pattern language with clearly described forces and trade-offs can be derived from existing development processes, and how the result can be used for a "Bottom-Up" approach to process adaptation.

Keywords
Extreme Programming, Agile Software Processes, Patterns, Methodology, Adapting, Constructing

CONSTRUCTING AGILE PROCESSES

In the past couple of years, *Extreme Programming* (XP) and other Agile Software Processes have started to gain considerable impact on software development in enterprise IT departments. Nevertheless, still many project managers are reluctant to switch to these new methodologies. They feel that some of the practices and techniques employed by Agile Processes are too alien to their companies' cultures. This is the reason why in many projects processes are not applied "by the book" but rather are adapted to the specific needs and standards of the environment.

However, existing Agile Processes usually do not specify explicitly how to be adapted. Aside from that, it is not easy to decide which of the growing multitude of Agile Processes to deploy in the first place. It requires much experience and a thorough understanding to select and tailor a new process to meet the specific requirements and constraints of a given software project.

The problems that need to be addressed by software processes are complex and cannot be solved without taking into account the concrete environment in which they are implemented. Pattern languages are an instrument to describe possible solutions to such complex problems in a structured way.

"[A] pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution." [Ale79, p. 247]

Pattern languages require the relationships between the patterns and the environment to be made explicit. Software process descriptions would benefit from this because the constraints for adapting them would be obvious.

Therefore, we believe it would be worthwhile to rewrite software processes as pattern languages. Moreover, if those could be integrated into a single comprehensive pattern language, we would have an alternative to *adapting* processes: Software processes could be *constructed*, using the concrete project-specific constraints as input.

Deriving the actual pattern languages for each of the Agile Software Processes is beyond the scope of this paper. Instead, we give some examples of patterns derived from different processes and show how they can be integrated, to become part of a common pattern language. Furthermore, we outline how such a common pattern language can be used to construct a custom process in a bottom-up fashion.

DERIVING THE PATTERNS

The foundation for the construction of custom software processes is a comprehensive pattern language. Its building blocks are patterns. Deriving them from existing processes suggests itself. An important criterion for the quality of a pattern is whether it has been applied successfully more than once. Therefore, we chose examples from software processes that have been proven in industry use.

The actual procedure of deriving patterns from a software process is interactive rather than formal. E.g. Christopher Alexander describes how patterns are conceived in an informal conversation (see [Ale79], p. 270ff). To the best of our knowledge, there does not even exist a formal procedure. We believe, professional judgement and discussion among peer is the only feasible foundation for such work. However, the focus of this paper is on the integration of patterns from different processes, and on constructing processes bottom-up. It is out of the scope of this paper to describe in detail how we arrived at the example patterns we use. This is not a trivial task, though, and will be a subject for other papers.

Software process literature is usually not already in pattern format. Where we find pattern-like structures, such as the core practices in Extreme Programming, they do not explicitly mention constraints and relationships to other elements. Other patterns can be found which are not primary elements of the process in question. E.g., Ex-

treme Programming requires the project team to work in a common office. This in itself is not one of the core practices, although it is understood by the XP community to be an important property of XP projects. In fact, a lot of discussions in newsgroups and on conferences concentrate on this topic: Is a common office necessary to do XP properly, and what if a common office is not feasible? Consequently, special care must be taken when deriving a pattern language from a process description to gather all the implicit patterns.When deriving a pattern language from a process, it is necessary to define a pattern structure first. In the following sections we describe and justify a possible structure that we found useful. Here is an example pattern we derived from XP:

Name: *Pair Programming*

Source: *Extreme Programming*

Problems:

Quality of Code: No software is free from defects. Reducing the number of defects can greatly improve customer satisfaction and maintenance costs.

Knowledge Distribution: During the course of a project, people acquire specific knowledge about the problem domain and the tools they use. Over several projects, this knowledge can be accumulated. However, people change jobs, they get sick, leave the company, go on holidays, or are generally not available when you need them.

Process Discipline: All software development processes and methodologies require discipline: Certain things have to be done, even if they are not particularly interesting. But most people are not good at self-discipline, especially when it is required over extended periods of time, like in a software project. If a task does not challenge them or seems too hard, people concentrate on quick ways to get over with it. Sloppiness creeps in, the defect rate rises, and the process is neglected. The same can happen when a project is close to some milestone and more work is left to do than the team is able to perform.

Constraints:

Readiness for Working in Pairs: Working in pairs even for a limited amount of time each day requires a certain readiness from the developers. Additionally, project management must not judge pair work as a waste of resources.

No Closed Groups: In order for distributing knowledge through pair work to succeed, it is necessary to mix pairs as much as possible. Closed groups of people who mix among themselves, but to not change between each other hinder distribution.

Implied Patterns:

One Location: Pair Programming requires the project team to work in one room, or at least in one building. Otherwise, the separate locations function as Closed Groups, see above.

Solution:

Pair Programming as described in [Bec00].

The example displays the general pattern structure that we use for collecting process patterns. The minimal set of features of a pattern are the *name*, the *problem* addressed, the *constraints*, that define the context in which it is applicable, and the *solution* to the problem. (See e.g. [Ale79], [G+95]) As can be seen in the example, our pattern structure adds some elements.

Different processes can use the same name for practices that are similar but still not identical. Therefore, when we derive a pattern we add the source process name to each pattern description.

We found that most patterns that can be derived from existing software processes actually address more than one problem at once. (Conversely, every problem is usually addressed by more than one pattern.) Therefore, the problem section of our pattern structure is a list of problems rather than a single problem.

The list of constraints is actually broken into two sections. The first is named "*constraints*" and lists the general preconditions and restrictions for the application of the pattern. The second is called "*implied patterns*". If one pattern requires another one to work, we call this an implied pattern. It is a real constraint, but of a special nature. We discuss the consequences of implied patterns in the section Constructing Processes below.

The *solution* part of the pattern often contains just a reference to the original process description. Only if the pattern is not obvious or not sufficiently described there, we add some explanation of the solution itself.

This pattern structure we found applicable to many different software processes. Below is an example from the *Capability Maturity Model* (CMM), a process model believed to be non-agile by many.

Name: Peer Review

Source: CMM for Software

Problems:

Quality of Code: See above, focus on defects

Process Discipline: See above

Implied Patterns:

Written Policy: All of the CMM practices require a written policy to be followed.

Constraints:

Time and resources need to be reserved for performing peer reviews, and for implementing their results, even under pressure. This requires support from management in critical situations.

Technical experts must be available for performing the technical part of the review.

Solution:

Peer Reviews as described in [P+93], p. L3-97ff.

Using this structure, each software process can be formulated as a pattern language. The individual languages (that describe one process each) will overlap, but they are still separate. In the following section, we will describe how they can be integrated into one common pattern language.

INTEGRATING THE PATTERN LANGUAGES

Each eligible software process will be refined into a separate pattern language. This in itself provides already valuable information about the patterns and their relationships therein. However, our goal is to provide a solid foundation for constructing agile processes. So the multitude of pattern language must be integrated into synthesis, a single, comprehensive pattern language.

One important source of differences between processes is the vocabulary. In order to unify them, it is therefore necessary to unify the vocabulary first. Since all software processes address a similar set of problems – after all it is always some kind of program that has to be developed and delivered – we suggest the collection of all problem descriptions from the pattern language in a catalogue.

This catalogue will be the starting point for process construction, as we will show in the next section, so particular care must be taken to make it complete and consistent. By complete we mean that all the problems addressed be a process must be included in the catalogue. By consistent we mean that problems should rather not overlap. Often, we find problems that are similar but not identical. It is then necessary to isolate the similarity into a new problem description to separate it from the two original problems.

Sometimes one of the original problem descriptions is then completely encompassed by the new one, while the other one is a superset. But equally often, we end up with three problems where before there were only two. If that happens, we have to go through all affected process descriptions and review them with respect to the new problem. Quite often, we find that patterns will have to be restructured as a result, and new patterns are found as a result. This is an iterative procedure that may have to be repeated for each of the problems of the respective pattern languages. It is tedious work, but it results in a much better (and much better documented!) understanding of the respective software processes.

When a complete and consistent problem catalogue has been built, the next step is to repeat the process of integration and unification for the pattern constraints and the patterns itself. This, however, is easier because there will be much less overlap. Certainly, processes use different terminology in these aspects as well. But since a lot of the constraints were formulated during pattern derivation rather then taken from the process descriptions, they are generally much more organized already.

Finally, we arrive at an integrated pattern language with the following characteristics: The problem descriptions are clearly formulated and separate. The constraints are unified and implied patterns are clearly marked. The patterns themselves define a common vocabulary. We further build three indices: 1) A list of the pattern names to be able to find the patterns in the catalogue; 2) a list of the pattern sources, to be able to find all patterns from one process; 3) a list of the addressed problems. The last index is the most important one for constructing processes but also the most difficult one to obtain. To create this index a list of key words must be build up, and all patterns must be harmonized, so that equal problems have equal problem entries in a pattern. To be able to construct a process from the catalogue we suggest to generate a dependency graph containing all filed patterns and showing the implied patterns.

CONSTRUCTING PROCESSES

Within most of the projects, processes are not applied out of the book. We conducted several interviews with project teams who claimed to use Extreme Programming. We found that most of them did employ certain techniques like *Unit Testing*, but left out others, like *Customer on Site*. This kind of adaptation was born out of necessity rather than ignorance: The circumstances simply did not allow to have a customer on site. Different projects also applied different changes to the process.

However, these adaptations are frequently done without control: First, the existing techniques within the process and beyond it are often not taken into concern, because they are not well understood. Second, the existing problems are not taken into account while adjusting a process to the specific concerns because there is no checklist or catalogue to compare against. Third, the dependencies between the chosen techniques are not clearly documented. This is where a pattern language helps. It can be used as a construction kit to build up a process.

To choose the building blocks of your process you first

have to think about the expected problems you will have to handle during the whole process. Most of the problems are very similar for almost every project and are already known. You can take them from the catalogue of problems in the pattern language. By integrating more than one process into one pattern language, even rare problems or such that are overlooked easily are provided in the form of a checklist from which you can select the problem areas that apply to a given project.

With the resulting collection of problems you can start selecting the patterns from the catalogue and build up your process. This is a creative work. Which patterns are actually chosen depends highly on the preferences, the existing technical and social skills of the people and several other circumstances of your project. The pattern language helps not to forget the options you have, and it helps not to lose the focus on your problems. It also helps ensure the consistency of the resulting process, by clearly documenting relationships between patterns, namely those that are implied by others. When all problem areas a covered by patterns, the process is complete.

The final question that remains is, in what way are the resulting processes agile? While it is beyond the scope of this paper to justify a formal definition of "agileness", we give one that covers what we find important in Agile Processes: 1) They stress importance of working software rather than that of process artefacts. 2) They consist of a minimum number of process elements. 3) They address the problem of rapidly changing requirements.

So on the one hand it is certainly possible to create "Big M" methodologies with a pattern language as the outlined in this paper. On the other hand, it does give project managers a tool with which they can create processes that consist only of a minimum number of patterns. When these patterns address the rapid requirements change problem (from the problem catalogue), then the resulting process should be reasonably agile.

ABOUT THE AUTHORS

Hasko Heinecke has been working in object-oriented software development since 1991. He has been the victim of both lack of process and too much process in various projects since then, and has been introducing Agile Process ideas and techniques to software projects since he worked with Kent Beck in 1999.

Christian Noack has been doing object-oriented software development for ten years. He has been working in the field of software processes within the last four years and is focussing on agile processes due to his work with Kent Beck in 1999 and Joseph Pelrine in 2000/2001.

Daniel Schweizer has a background of 9 years in software development and about 3 years with object technology. Since 2001, he has authored and taught courses in Smalltalk, and XP.

All the authors work for branches of Daedalos International.

REFERENCES

[AIS77] Ch. Alexander and S. Ishakawa and M. Silverstein: A Pattern Language, Oxford University Press, New York, 1977

[Ale79] Ch. Alexander, The Timeless Way of Building, Oxford University Press, New York, 1979

[Amb98a] S.W. Ambler: Process Patterns: Building Large Scale Systems Using Object-Oriented Technology, SIGS Book, Cambridge University Press, New York, 1998

[Amb98b] S.W. Ambler: More Process Patterns: Delivering Large Scale Systems Using Object-Oriented Technology, SIGS Book, Cambridge University Press, New York, 1998

[Bec00] K. Beck, Extreme Programming Explained: Embrace Change, 2000, Addison-Wesley

[Cop95] J.O. Coplien: A Generative Development-Process Pattern Language, Pattern Languages of Program Design, 1995, Addison Wesley Longman, Inc., pp. 183-237

[Cun96] W. Cunningham: EPSIODES: A Pattern Language of Competitive Development, Pattern Languages of Program Design 2, 1996, Addison-Wesley, pp. 371-388.

[G+95] E. Gamma and R. Helm and R. Johnson and J. Vlissides: Design Patterns, 1995, Addison Wesley

[P+93] M.C. Paulk et al.: Key Practices of the Capability Maturity Model, Version 1.1