

## 18. Defining Agile Patterns

*Teodora Bozheva, Maria Elisa Gallo*

**Abstract:** The variety of agile methods and their similarity could be a problem for software engineers to select a single or a number of methods and to properly utilize them in a project. An approach to resolving it is to provide concise and adjustable solutions of problems, recurring under certain circumstances, with justification of why and how to apply them. In this chapter we present an approach to acquiring and defining knowledge about agile software development in terms of patterns. We emphasize the rationale in the pattern structure. We discuss how the usage of the agile patterns contributes to organizing and delivering organizational knowledge and to improving the software processes in an organization. Early results from industrial trials are presented to demonstrate additional benefits, which an organizations gains from adopting the agile patterns. In the concluding part we define the directions for further research on the topic.

**Keywords:** Agile methods, Pattern, Rationale Representation, Knowledge organization and delivery, Software process improvement

### 18.1. Introduction

Software systems evolve from a large number of decisions taken over an extended period of time. Nowadays, there are plenty of methods, maturity models, and body-of-knowledge books explaining or providing guidelines on how to organize and perform software engineering and management activities. However, due to the continuous rush for improving business results, companies are usually interested in adopting methods which provide flexibility with respect to practice implementation and improvement.

Agile methods recognize that any project, team and organization have their unique peculiarities. Therefore, instead of trying to unify the approaches to developing software, these methods respond to the specific needs via business value based prioritization, short feedback cycles and quality-focused development. When appropriately applied, the agile practices bring a number of business benefits such as better project adaptability and reaction to changes, reduced production costs, improved product quality and increased user satisfaction with the final solution. A good overview of the agile methods is provided in [10].

Different factors, however, determine the need and the success of implementing agile practices in an organization: team size, product criticality, project dynamism, personnel, and skills. In [3] B. Boehm and R. Turner define them as agile method home grounds. Although there are plenty of publications on the agile methods, the question how to combine single practices from different methods to define an organization's specific process and when such combinations are reasonable, still remains unclear. This might be the reason, for which lots of people find the agile methods rather unconvincing.

A pattern describes a solution to a recurring problem in such a way that the solution can be used multiple times without being done the same way twice. In general, a pattern has three essential elements: (1) problem - situations, in which the pattern is appropriate to be applied; (2) solution - activities which the pattern consists of; (3) consequences - results and trade-offs of applying the pattern. The solution is abstract enough to make it possible to apply it in different situations.

The usage of patterns for organizing re-usable knowledge is not new in the software engineering field. Two widely known applications of it are described in [4] and [8]. Patterns provide a means for the organizations to build processes, which fully correspond to their project and organizational contexts; like building a house of Lego parts. From process improvement point of view, since each pattern addresses a specific problem, it can be easily tried out and adapted appropriately before being put in place in a project or in the whole organization.

Emphasizing rationale in a pattern that defines a software engineering activity facilitates the understanding of when and how to implement the pattern, and contributes to consistent rationale documentation and usage in an organization.

An agile pattern is a pattern, which is based on agile methods. This means that the solution to a problem uses practice(s) from one or several agile methods for software development. In addition an agile pattern includes rationale for applying the solution in a specific context. That is an agile pattern extends the classic pattern definition with providing guidelines on how to implement the pattern activities in different situations.

The agile patterns discussed herein are derived from the following agile methodologies eXtreme Programming (XP) [1,2], Scrum [14], Feature Driven Development (FDD) [12], Lean Development (LD) [13], Adaptive Software Development (ASD) [9], Agile Modeling [1]. The work on the patterns definition has been initiated within the Framework of Agile Patterns project (S-OD03ES07), partially funded by the Basque Government. The main goal for the project was to contribute to the adoption of the agile

methods by software intensive organizations by defining a framework of agile patterns, which can be easily deployed and adapted to the specific needs of an organization.

The results, which we present in this chapter, address two main issues:

- Defining agile patterns: derive and recover best practices from agile methods and emphasizing the rationale in the pattern structure;
- Using agile patterns for spreading knowledge through an organization and for software process improvement.

We present our motivation for defining the framework of agile patterns in Sect. 18.2. In Sect. 18.3 we discuss how we derive knowledge from the lightweight methods and how we structure it in the pattern template. Then we discuss the potential benefits from using the agile patterns from rationale management point of view (Sect. 18.4). In the final section we present open issues for further research.

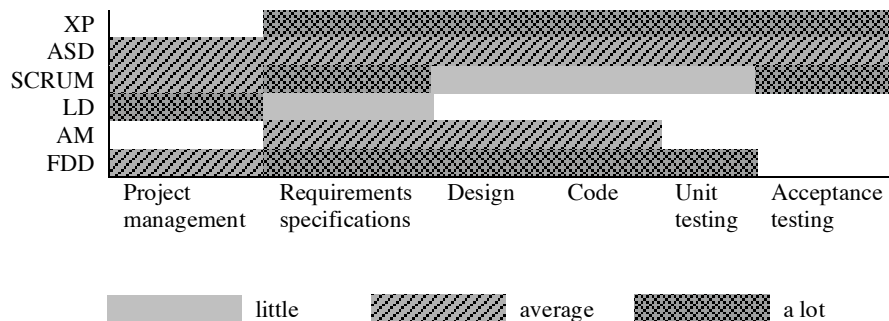
## 18.2. Motivation for Defining Agile Patterns

There are a number of agile methods, which propose different approaches to software development and management. Although the individual practices can vary, all the methods propose maintaining good understanding of the project objectives, scope and constraints among the team members, developing software in short, feature-driven, customer-relevant iterations, receiving constant feedback from the customer and the developers, and focusing on the delivery of business value.

Some agile approaches focus more heavily on project management and collaboration practices, e.g. LD, ASD, and Scrum. Others such as XP, FDD and AM focus on software implementation practices (see Fig. 18.1).

Two main principles guided us when defining the Agile Patterns:

- From software engineering point of view: Provide benefits to the people involved in software development by defining a set of appropriate activities and alternatives of them in an agile manner, i.e. easy to understand and apply and flexible with respect to combining several patterns.
- From knowledge management perspective: Support the use of agile methods by providing rationale for implementing a pattern in terms of guidelines for selecting and executing appropriate activities, which address a specific problem. In this way, the rationale will be immediately available to the roles who execute the pattern at the time they need it.



**Fig. 18.1.** Number of Agile Practices Related to a Software Development Process

We have focused on defining patterns related to

- Software engineering practices
- Project management practices, and
- Practices related to customer involvement and collaboration.

Software engineering and project management practices are the basic ones to be put in place to ensure effective, adaptive and easy to implement software development process. Customer involvement and collaboration is one of the key factors for success of a software development project. Therefore, we have decided to start working on these three pattern categories first.

Each agile pattern addresses a very concrete issue, e.g. *Increase the feedback from the development team to the management (Development-FeedbackIncraser)*. It describes activities to be performed to accomplish the issue, roles involved in executing the activities, and resulting products. Wherever possible, alternative activities are considered. The pattern also includes a piece of process rationale related to selecting an alternative solution to the addressed problem. The pattern structure is explained in the next section. The idea is to select patterns taking into account the characteristics of the work to be done and the context, in which it is going to be carried out, and to use them to organize and develop a project.

Several reasons made us decide to integrate rationale in the pattern structure:

- The patterns describe activities that are typically performed by software engineers and project managers who are accustomed to using well-structured information like pattern definitions.
- Using an already known and easy to read structure for rationale representation motivates rationale capturing and usage.

- As long as the rationale related to a specific problem is represented together with the approach to resolving the problem and the context, in which the solution works, it is easier for the implementers to decide upon the activities they have to perform to address the problem.
- If several patterns are put together to form a process, the combination of the rationale related to each pattern will provide to a great extent the rationale for the whole process.
- The pattern descriptions are general enough to be used in different projects. Rationale captured in real-life projects carried out in an organization can be added into the pattern definitions. That is the patterns support sharing organizational knowledge across multiple projects.

In addition patterns could be used for representing rationale related to other types of activities, different from software engineering, e.g. contract management activities. In general, keeping organizational information consistent and in a common format supports its usage and maintenance.

### 18.3. Agile pattern definition approach

A pattern describes a problem, which typically occurs under certain circumstances. It describes a basic approach to solve a problem providing opportunities to adapt the solution to the particular problem context. The three essential elements of a pattern are problem, solution and consequences.

#### 18.3.1. Pattern types

Three key terms take part in the agile methods descriptions: *practices*, *concepts* and *principles*. *Practices* describe specific actions that are performed in the whole process of software development, e.g. create product backlog (SCRUM). *Concepts* describe the attributes of an item, e.g. a project plan. *Principles* are fundamental guidelines concerning software development activities, e.g. empower the team (LD).

Each practice can be described by pattern with the following attributes:

- **Intent:** a short description of what the objective is;
- **Origin:** methodologies, from which the pattern originates;
- **Category** to which the pattern belongs. With respect to the type of issues addressed, the patterns are grouped in the following

categories: *Project and Requirements Management, Design, Implementation and Testing, Resource Management, Contract Management and Software Process Improvement.*

- **Application scenario:** context, in which the pattern is appropriately applied;
- **Roles:** people involved in carrying out the pattern and their responsibilities;
- Main and alternative **Activities** that constitute the pattern. Activities can invoke other patterns;
- **Tools** that support the pattern execution;
- **Guidelines** for performing the activities including suggestions for making a decision about which alternative solution to choose when.

This structure is closest to the one used in [8]. Sect. 18.3.3 discusses in details the definition of practice patterns.

In the attempt to formalize the definitions of the *concepts* and the *principles*, we found out that the only difference from the *practice* definition by means of patterns is that the *concepts* and the *principles* do not include the Activities attribute. Therefore, we decided to handle *concepts* and *principles* as *practice* patterns, i.e. with nearly the same structure, but with different content.

*ProjectPlan* and *CollectiveCodeOwnership* are examples of a concept and a principle pattern respectively. We include them here to illustrate the commonalities and the differences between the structures of the definitions of the three terms.

#### Concept pattern: *ProjectPlan*

**Intent:** Serves as a focal point and quick reminder of the most important elements about the project.

**Origin:** ASD: Project Data Sheet

XP: Release plan

FDD: Development plan

**Application scenario:** Project planning

**Roles:** Customer: makes business decision (scope, priorities, release planning)

Developers: make technical decisions (effort estimations, risks)

Project Leader: makes the planning

**Definition:** The Project Plan is one-page summary of the key information about the project. The Project Plan includes the following details:

- Project objectives statement
- Overall Architecture
- Major project milestones
- Core team members

The project objectives statement should be specific and short (25 words or less), and it should include important scope, schedule, and resource information.

**Guidelines:** In FDD the development plan consists of:

- Feature sets with completion dates
- Major feature sets with completion dates derived from the last completion date of their respective feature sets
- Chief Programmers assigned to feature sets
- The list of classes and the developers that own them

**Principle pattern: *CollectiveCodeOwnership***

**Intent:** The code is collectively owned by the developers. Anyone can do changes to the code. The programmers use a coding standard to enforce a common style.

**Origin:** XP: Collective Code Ownership

**Guidelines:** Collective code ownership is more reliable than putting a single person in charge of watching specific pieces of code, especially because, if a person leaves a project at some time, the other project team members will know the code he has implemented and will be ready to continue his work.

### 18.3.2. Agile patterns from design rationale perspective

Let us consider the practice pattern definition structure as the most complete one.

Compared to the classic pattern definition (problem-solution-consequences), Intent and Application scenario correspond to the problem attribute. Activities matches to solution. Some patterns provide alternative solutions to the same problem. This typically happens when the problem is addressed by more than one agile method and different solutions to it are proposed. Guidelines include hints for performing the activities and the consequences from them. The Guideline in the *CollectiveCodeOwnership* principle is an example of a consequence from applying a pattern.

A complete definition of an effective process for resolving a particular issue includes activities that address the problem, people with relevant

skills and knowledge and tools supporting the process implementation. Therefore, to provide complete information about resolving a problem, we have added the Roles and Tools attributes.

To show how the rationale related to making a solution to a problem is included in the pattern itself, let us compare the agile pattern scheme with the Question-Option-Criteria (QOC) [6] [Chap. 1, Sect. 1.3 in this book] one for argumentative design rationale. Intent and Application scenario correspond to the QOC Question element. Activities matches to the Options element. The Guidelines attribute of the agile pattern includes Criteria and Arguments for choosing an option. Wherever relevant, Guidelines also provide explanations about the relationship between Options and Criteria that is it includes the QOC Assessment element.

For instance, the *CodeImplementer* pattern, discussed in more details bellow, has intent “Implement code”, which corresponds to the question “How do I implement code?” Two possible activities are defined for addressing the intent, namely apply the *FDDCoder* pattern or the *XPCoder* one. *FDDCoder* describes how to implement code following Feature Driven Development, while *XPCoder* explains how to do coding applying eXtreme Programming. These alternative activities match to the Options element of the QOC scheme.

Applying *XPCoder* requires a tool supporting test-driven-development to be used by the developers. That is the QOC assessment element “To use *XPCoder* you need a tool supporting test-driven development” is part of the Guidelines attribute of *CodeImplementer*.


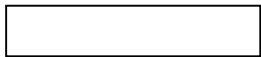
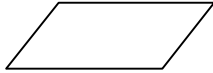

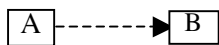
Following the classification of design rationale approaches, made in [6] [Chap. 1, Sect. 1.2 in this book], the method we present herein can be characterized as prescriptive and less-intrusive. Although a predefined scheme for rationale representation is used, we categorize it as less-intrusive, because the scheme can be easily adapted to an organization’s specific needs. Besides it is not required that all the pattern elements be defined at once. This scheme can be used for defining rationale from earlier to most elaborated stages.

### **18.3.3. Defining agile patterns: an example**

The patters, at the time of writing this book chapter, address software engineering activities from the Engineering, Management and Reuse process areas from the SPICE process standard [15]. The patterns have been derived from the agile methods mentioned above.

Apart from the natural language description of the patterns, every category is graphically illustrated showing which patterns, concepts and principles it includes, and the relationships between them.

On the graphics the following symbols are used:

Symbol	Meaning
	Principle
	Pattern
	Concept
	<i>“invokes”</i>
	Pattern A supports B, but it is optional to use A when implementing B

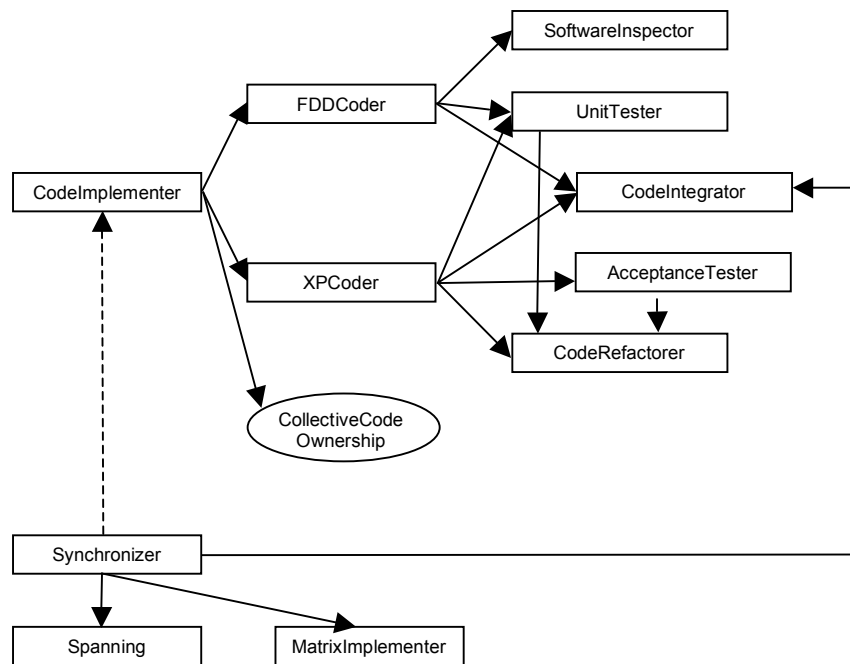
**Fig. 18.2.** Symbols used in the pattern diagrams.

As an example, Fig. 18.2 shows the diagram for the Implementation & Testing category.

We have studied the agile methods to determine to which phases of the software development process they are applicable. Since some methods consider the same issues (see Fig. 18.1), but provide different approaches to resolving them, the main difficulty was to identify the similarities and differences between the solutions and to decide when to use the activities proposed by each one of the methods. The granularity of the patterns was another debatable issue. We used our experience with the agile methodologies in defining the patterns content. However, we have realized that the decision-making criteria have to be explicitly specified and this is one direction of our future activities.

To define a pattern we grouped activities from different agile methods that had similar objectives and then we defined the pattern structure. Afterwards, we established the relationships among all the activities proposed in the different agile methods together with the reasons for selecting one over another activity, wherever alternatives were available. Then we identified which atomic practices formed a pattern, what alternatives were pos-

sible and for each alternative what made it feasible. We added the intent, the roles, the tools and the guidelines to the pattern structure as well.



**Fig. 18.3.** Implementation & Testing category

The result was a group of patterns strongly related to each other. In this group there are patterns that invoke other patterns. This happens when a pattern contains a group of activities that can be used in other patterns as well. Therefore, one pattern can be shared across several other patterns. In the following example we can see such relationships between patterns in the Implementation & Testing category.

Let us have a look at the *FDDCoder* and the *XPCoder* patterns.

**Pattern: FDD Coder**

**Intent:** Implement defect-free code

**Origin:** FDD: Build by feature

**Category:** Implementation & Testing

**Application scenario:** A developer implements a piece of code.

**Roles:** Developers: write Unit Tests, implement and integrate the code;  
Customers write acceptance tests.

**Activities**

1. *Implement code for a feature.*
2. *Apply Unit Tester.*
3. *Apply Code Integrator.*
4. *Run test cases and improve the code until all test cases pass. The cycle finishes when everything that could possibly break is tested. At the end of a feature implementation apply Software Inspector.*

**Tools:** Reference to Testing supporting tools is provided in the original document.

**Guidelines:** No specific guidelines to the implementation of this pattern are defined yet.

**Pattern: XP Coder**

**Intent:** Implement defect-free software

**Origin:** XP: Coding

**Category:** Implementation & Testing

**Application scenario:** A developer implements a piece of a software product (a feature)

**Roles:** Developers: write Unit Tests, implement and integrate code;  
Customers write acceptance tests.

**Activities**

1. *Apply Unit Tester and implement code consecutively until a feature is implemented*
2. *Apply Code Integrator.*
3. *Apply Unit Tester.*
4. *At the end of the day (or at the end of an iteration) apply Acceptance Tester.*
5. *Apply CodeRefactorer.*

**Tools:** Reference to tools supporting test-driven development is provided in the original document.

**Guidelines:**

1. Apply Pair Programming (Two developers, one keyboard) for higher quality, faster development, less defects and more fun during the implementation process
2. Apply CollectiveCodeOwnership to the code, i.e. all the team members may modify the whole code of the product.

3. Use a coding standard to ensure a common style of implementation and that everyone can read and understand any code in the system
4. Good names substitute comments
5. Express intention, not implementation
6. Use a coding standard to facilitate the modification of the code by any team member (XP: *collective code ownership*).

The underlined activities are patterns that are invoked by the *FDDCoder* and the *XPCoder* patterns. This means that to perform *XPCoder*, for instance, we need to perform the patterns *UnitTester*, *CodeIntegrator*, *AcceptanceTester* and *CodeRefactorer*.

*FDDCoder* and *XPCoder* have two patterns in common: *UnitTester* and *CodeIntegrator*. However, in *FDDCoder* *UnitTester* and *CodeIntegrator* are performed after the implementation of the code, while in *XPCoder* *UnitTester* is the first practice to perform. In XP the developers first have to create a unit test framework to be able to define automated unit tests suites. All the tests must be created before the actual code that implements the test. The main reason for this is to keep writing code, which surely meets the software requirements and is always test-proven.

Fig. 18.3 shows the relationships among the patterns that take part in the definitions of *FDDCoder*, the ones that take part in the *XPCoder* and the links between all the patterns and principles involved in the Implementation & Testing category. There are no concepts in this category.

The main pattern, ***CodeImplementer***, defines the alternatives to develop the code: *FDDCoder* and *XPCoder*.

If *FDDCoder* is selected, then it invokes other patterns that include the activities needed to develop the code in the way FDD proposes. These patterns are *SoftwareInspector*, *UnitTester* and *CodeIntegrator*. The complete group of activities included in these patterns tells us how to perform the Code following the FDD method.

If *X Coder* is selected, the patterns invoked are *UnitTester*, *CodeIntegrator*, *AcceptanceTester* and *CodeRefactorer*.

*CodeImplementer* also invokes the *CollectiveCodeOwnership* principle. This means that for the development of the code, one important consideration to take into account is the ownership of the code.

There are other “invoke” relationships between patterns in this category: *UnitTester* and *AcceptanceTester* invoke *CodeRefactorer*. This denotes that when implementing these patterns it is necessary to perform the activities included in *CodeRefactorer*.

Another pattern, *Synchronizer*, takes part in this category. It invokes *Spanning*, *Matrix Implementer* and *CodeIntegrator* as different alternatives

to perform the synchronization between code, generated by several people. This means that *CodeIntegrator* is a pattern shared by several patterns in this category. The dotted arrow from *Synchronizer* to *CodeImplementer* shows that they are related, more precisely, *CodeImplementer* supports the implementation of *Synchronizer*, but *Synchronizer* does not explicitly invoke *CodeImplementer*.

A drawback of the diagrams, which we currently use, is that they do not explicitly show if some activities are alternative to each other or have to be executed collectively. In particular, in the Implementation & Testing category *FDDCoder* and *XPCoder* are alternative approaches (options) for implementing *CodeImplementer*. The same is true for *Spanning* and *MatrixImplementer* invoked by the *Synchronizer* pattern. However, this is clearly stated in the textual patterns definitions. For instance the *CodeImplementer* definition looks like this:

**Pattern: *CodeImplementer***

**Intent:** Implement code

**Origin:** FDD: Build by feature  
XP: Coding

**Category:** Implementation & Testing

**Application scenario:** Developer implements a piece of code

**Roles:** Developers

**Activities**

Alternatives are:

- *FDDCoder*
- *XPCoder*

**Tools:** see Guidelines.

**Guidelines:**

- Applying *XPCoder* requires using a tool supporting test-driven development
- Automated tests can save hundred times the cost to create the tests themselves by finding and guarding against bugs. The practice of using automated tests shows that the harder it is to write a test, the more it is needed and the greater the savings will be.
- Consider applying the *CollectiveCodeOwnership* principle.
- Any place where several individuals are working on the same thing, a need for synchronization occurs. Refer to *Synchronizer* for putting together and maintaining code implemented by different developers.

## 18.4. Using the agile patterns

Nowadays lots of organizations face the need to adapt quickly to modifications requested by their customers, changes on the market or challenges from competitors. This happens in small as well as in large organizations, in disciplined certified (ISO 9001:2000, CMMI®<sup>1</sup>) companies as well as in ones that follow their internal development processes. Organizations that address these problems need to acquire, apply and extend the knowledge related to some or to all the aspects of software engineering and management.

The knowledge represented by means of the agile patterns addresses the key software development and process improvement activities. The patterns are easy to understand, neither require tool support for modifications, nor a special methodology to maintain them. They can be adjusted to the needs of each organization.

From design rationale point of view the main usage of the patterns is to facilitate the knowledge transfer, in particular the organization and delivery of reusable knowledge within an organization, as well as to support learning from the past and on-the-job training.

From software engineering perspective the activities that get most benefits from capturing rationale in terms of agile patterns are the engineering (from requirements elicitation to system testing) and the process improvement ones. Since the convenience of using patterns to define software engineering activities is obvious, bellow we are going to discuss only how the patterns support software process improvement.

These two viewpoints are completely aligned with the design rationale uses described in [6] [Chap. 1, Sect. 1.4 in this book] and are briefly discussed bellow.

At the time being a web repository of the agile patterns is being developed. It will provide possibilities for finding patterns and seeing their relationships with other patterns in the framework.

### 18.4.1. Supporting knowledge transfer

The agile patterns, at their current stage, are primarily focused on organizing knowledge about performing specific software engineering activities. The patterns are described in natural language and use a simple format.

---

<sup>1</sup> Capability Maturity Model Integrated, developed by Software Engineering Institute, Carnegie Mellon University, <http://www.sei.cmu.edu>

Therefore, it is very easy to maintain them and to add newly acquired knowledge.

The benefits for a company using the present state of the framework of agile patterns are that

- The framework structure can be easily adapted to reflect the way the software engineering activities are performed in the organization;
- The patterns can be enhanced with knowledge acquired by the software engineers in the company.

This increases the value of the framework for the organization since the knowledge presented in the patterns reflects the experience and the culture of the same organization. Moreover, the pieces of rationale, which the patterns include, typically address concerns how to approach a specific problem and what alternative solution to select when. That is, knowledge acquired by software engineers in the past is captured and available for delivery through the organization.

Improving the pattern scheme as to more explicitly define rationale related to a pattern problem, would additionally boost the know-how transfer within an organization.

Since the patterns focus on particular activities, it is easy to explain, understand and apply single ones of them. This significantly supports the on-the-job training of people. Besides, the application of the agile patterns implies lots of team work and collaboration that additionally facilitates the spreading of the available knowledge.

#### **18.4.2. Supporting Process Improvement**

People having experience with adoption of new approaches to software development know that the Big Bang style of implementation of new processes hides a number of potential drawbacks. Some of them are related to the risk that the new processes as a whole are only partially understood by the people who have to apply them due to the inherent complexity of the process architecture. At the same time difficulties are often faced when trying to split a process to smaller elements (steps) in order to focus on improving the performance of a particular element only. The agile patterns support the process improvement activities exactly by providing a means to pilot single process elements (patterns) before integrating them into an entire process.

Since the agile methods are all oriented towards rapid achievement of business goals, the successful implementation of the agile practices re-

quires considering additional factors like personnel experience, organizational culture, and size and criticality of the projects, in which the practices are applied. That is, apart from the pure engineering activities to be performed, additional issues determine the successful application of the pattern in a specific context. The agile patterns include a wide spectrum of rationale issues related to the integration of a specific pattern in a process built of other patterns. For instance, the *CollectiveCodeOwnership* pattern implies that all the team members have access to and are allowed to modify the code of the product implemented by the team (organizational culture issue).

### 18.4.3. Industrial usage

The definition of the agile patterns has been done on two steps: (1) define their structure and content based on study of the literature about the agile methodologies and (2) enrich them with practical experience gathered in trials of the agile methods and/or the agile patterns themselves. The second step can be considered as both piloting and continuous refinement of the patterns. It is not mandatory that the second step begins only when the first one is finished, because presenting an agile practice, known from the literature, by means of a pattern does not make much sense for the software engineers. However, when the description of the practice is complemented with rationale related to its implementation, it brings much more value.

We started piloting the patterns in parallel to defining them. Valuable input came from seven projects, which were focused on experimenting XP and PSP (Personal Software Process) practices in e-commerce and e-business application development. The trials were carried out within the *eXpert* project (IST-2001-34488) [7], partially funded by the European Commission. They were performed by teams in different organizations, located in Spain, Germany and Bulgaria. The main objectives for the trials were to study how the agile practices contribute to increasing the productivity and the efficiency of the software engineers, and to improving the quality of the products they develop<sup>2</sup>.

From design rationale perspective, our main goal was to find out underlying principles that would help organizations to implement the agile pat-

---

<sup>2</sup> For the sake of completeness, the results from the experiments are as follows: Productivity increased up to 73%. One company decreased its productivity; Schedule deviation reduced between 7% and 38%; Cost deviation decreased up to 31%. Only one company increased its cost deviation; Defect rates reduced between 10% and 83%.

terns. The observations and the findings of the trials were used to refine the agile pattern definitions.

With respect to the patterns adoption the pilot projects in two of the companies showed that introduction of agile practices has to be made gradually. First, organizations have to select the process, whose agility they aim to increase. Then the patterns that could be used to improve activities from these processes have to be identified and piloted in order to be adjusted to the practices, which are currently in place in the organization. That is, the focus should be on a small set of activities and the patterns that affect them.

Three teams studied the communication between the development team and the customer (*FeedbackIncraser*). In none of the teams the customer was on-site as recommended by XP. However, agreements were made that the customer would clarify developers' doubts and questions by means of regularly reviewing the current product status and providing feedback by email, phone or direct conversations. One of the teams tried the "developer-on-site" alternative, which consists of periodically sending a developer to the customer's office to demonstrate the product and collect feedback. All the developers recon that the improved communication with the customer had a positive feedback on the decisions made in the project with respect to what features to be implemented, how and when.

The rationale, acquired during the experiments, with respect to how to resolve specific problems, is documented in the patterns themselves. However, we realized that the approach of representing software engineering knowledge in terms of patterns will benefit significantly from improving the pattern structure as to better represent the design rationale related to the resolution of a particular problem. Yet, an important condition is that the patterns remain easy to maintain, use and adapt to organizational needs.

## 18.5. Conclusions

Formalizing knowledge is a costly process. Aiming at achieving a perfect formalization is perhaps not worth, because software development, as any other intensive human activity, is evolving. Therefore the focus should be on providing an easy to customize and simple to apply solutions like the framework of patterns. Then define criteria for making decisions on how to adapt a pattern to a particular context, why to choose a practice over another one, an option over another possibility, and so on. Enriching the framework with worst apart from best practices is also considered useful.

However, it is most probable to be performed only for the internal needs of the organization.

The main directions of future work on the subject include improving the structure of the patterns to better organize different types of rationale associated with the problem, which a pattern addresses; investigating approaches for adopting the framework of agile patterns; studying the benefits an organization gains due to capturing rationale in terms of patterns and exploiting it.

## References

- [1] Ambler S, Jeffries R (2002) *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. John Wiley and sons
- [2] Beck K (2000) *Extreme Programming Explained: Embrace Change*. Addison-Wesley
- [3] Boehm B, Turner R (2003) *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley
- [4] Coplien J, Douglas C. Schmidt (1995) *Pattern Languages of Program Design*. Addison Wesley.
- [5] Dutoit A, Paech B (2002) *Rationale Management in Software Engineering, Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing Company
- [6] Dutoit AH, McCall R, Mistrík I, Paech B (2006) *Rationale management in software engineering*. Heidelberg: Springer-Verlag
- [7] eXpert project. <http://www.esi.es/Expert>
- [8] Gamma E, et al (1995) *Design Patterns*. Addison-Wesley
- [9] Highsmith JA (2000) *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing
- [10] <http://www.agilealliance.org/>
- [11] <http://www.dsdm.org>
- [12] Palmer S, Felsing J (2002) *A Practical Guide to Feature-Driven Development*, Prentice Hall
- [13] Poppendieck M, Poppendieck T (2003) *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley
- [14] Schwaber K, Beedle M (2002) *Agile Software development with Scrum*. Ambler SW (2002) *Agile Modelling*. John Wiley
- [15] SPICE process standard. <http://www.sqi.gu.edu.au/spice/>