2

Agile and UML-Based Methodology

The most valuable insights are the methods.
Friedrich Wilhelm Nietzsche

As useful modeling language must be embedded in a methodology. This chapter presents characteristics of agile methods, in particular those of the process of *Extreme Programming (XP)* [Bec04, Rum01]. Using these characteristics and further elements, the chapter introduces a proposal for an agile methodology based on UML.

B. Rumpe, *Agile Modeling with UML*,
DOI 10.1007/978-3-319-58862-9_2

For many years now, the improvements in Software Engineering have been responsible for the continual increases in efficiency in software development projects. These days, software has to be created with a high level of quality with increasingly fewer personnel resources in increasingly shorter timeframes. Processes such as the Rational Unified Process (RUP) [Kru03] or the V-Modell XT [HH08] are more suitable for large projects with a lot of team members. For smaller projects, however, these processes are overloaded with activities that are not entirely essential for the end result. Consequently, the processes are too inflexible to be able to respond to the rapidly changing environment (technology, requirements, competing products) of a software development project.

A relatively new group of processes has arisen under the common label of "agile processes". These new processes are characterized primarily by flexibility and quick feedback. They concentrate on the main work results and focus on the people involved in the project—in particular, customers.

The Standish Report [Gro15] describes how significant causes of project failure can be found in poor project management: there is insufficient communication; too much, too little, or incorrect documentation; risks are not counteracted in time; and feedback is requested from users too late.

It is the human element in particular—the communication within the project and the cooperation amongst the developers and between the developers and the users—that is seen as the main cause of failure for software development projects. [Coc06] also states that projects rarely fail for technical reasons. On the one hand, this indicates that the developers have a good command of even innovative technologies; but on the other hand, this claim must be questioned to some extent at least. If the technology used causes difficulties, these problems often disrupt the communication between team members. As the project continues, these problems in communication come to the fore for emotional reasons and are ultimately remembered as "perceived" reasons for the failure of the project.

Because use only a reduced set of appropriate languages and tools is used this generation of processes can be regarded as lightweight. The more compact the language and the better the analysis and generation tools, the less redundancy is necessary. In addition, the efficiency of the developers is improved by reducing additional efforts and expenses for management and documentation.

Today, Software Engineering offers an extensive *portfolio* of approaches, principles, development practices, tools, and notations that we can use to develop software systems in various forms, sizes, and levels of quality. These elements of the portfolio complement each other to some extent, but can also be used as alternatives to one another, which means that there is a wide range of selection options available for managing, controlling, and executing a project.

This chapter looks firstly at the current status of the Software Engineering portfolio. Section 2.2 discusses "Extreme Programming" (XP) and Section 2.3

presents three essential practices from XP. Section 2.4 contains a sketch for a method that is suitable as a reference for the use of UML/P and the techniques discussed in detail in this book.

Chapter 3 provides a compact overview of the Unified Modeling Language profile UML/P. This profile can be used for the method proposed here and is supported, e.g., in [Sch12] with a suitable tool. Like UML itself, UML/P is largely *not method-specific*. This means that it is possible and even helpful to use UML/P in other methods. However, as UML/P focuses on the ability to generate code and tests, it is particularly suitable for agile methods.

2.1 The Software Engineering Portfolio

[AMB+04] and [BDA+99] attempt to consolidate the knowledge about Software Engineering that has built up over more than 40 years into a "Software Engineering Body of Knowledge" (SWEBOK). In the SWEBOK, concept formations are standardized, the main core elements of Software Engineering are presented as an engineering discipline. The goal is to establish a generally accepted consensus about the content and concepts of Software Engineering.

Some of the terminology used for software development processes which is significant for our deliberations is shown in Fig. 2.1.

Experience gained from the execution of software development projects in recent years clearly shows that there cannot be *one, unique* process for software development: Projects strongly differ in importance, size, area of application, and project environment. Instead, efforts are being made to compile a collection of concepts, best practices, and tools that allow project-specific requirements to be taken into account in an individual process. The level of detail and the precision of the documents, milestones, and results to be delivered are defined dependent on the size of the project and the desired quality of the results in each case. Existing process descriptions, which can be viewed as templates, can be helpful. However, project-specific adjustments are considered necessary in most cases. Therefore, it is useful for those involved in a project to be familiar with as many approaches from the current portfolio as possible.

The 1990s saw a strong trend towards complete and therefore rather bureaucratic software development processes. The agile methods of the 2000s have broken away from this trend. Two factors enabled this change of direction: firstly, the significantly increased understanding of the tasks involved in developing complex software systems; and secondly, the availability of improved programming languages, compilers, and a number of other development tools. Today, it is almost as efficient to implement a GUI immediately as it is to specify the GUI. The specification can therefore be replaced by a prototype which the user can try out and which can be reused in the realization of the final product. The trend towards reducing the level of required

Development method: A development method (synonym *process model*) describes the procedure "for executing software creation in a project" [Pae00]. We can differentiate between a technical, a social, and an organizational development method.

Software development process: This term is occasionally used as a synonym for "development method" but is often understood as a more detailed form. Thus, [Som10] defines a process as a set of activities and results used to create a software product. In most cases, the chronological sequence or the dependencies of the activities are also defined.

Development task: A process is divided into a series of development tasks. Each task delivers certain results in the form of *artifacts*. The team members participating in the project perform *activities* in order to complete these tasks.

Principle: Principles are fundamentals on which action is based. Principles are generally valid, abstract, and as general as possible in nature. They form a theoretical basis. Principles are derived from experience and findings (see [Bal00]).

Best practices: This term describes successfully tested *development practices* in development processes. A development practice can be perceived as a specific, operationalized process pattern that implements a general principle (see RUP [Kru03] or XP [Bec04]).

Artifact: Development results are represented with a specific form of notation—for example, natural language, UML, or a programming language. The documents of these languages are called *artifacts* and examples include requirements analyses, models, code, review results, or a glossary. An artifact can have a hierarchical structure.

Transformation: Developing a new artifact and improving a version of an existing artifact can both be understood as transformations. The development or improvement can be automated or manual. Ultimately, almost all activities can be seen as transformations of the set of given artifacts in a project.

Fig. 2.1. Terminology definitions for the software development process

developer capacities is intensified by the fact that having fewer people involved in a project also reduces the level of organizational overhead, which in turn can further reduce the workload.

When we use agile methods, the increased emphasis on the individual capabilities and needs of the developers and customers involved in a project allows us to reduce project bureaucracy even further in favor of greater individual responsibility. This focus on the team can also be observed in other areas of economic life—for example, where flat management hierarchies are used. It is based on the assumption that mature and motivated project participants will demonstrate responsibility and courage by taking the initiative when the project environment gives them the opportunity to do so.

2.2 Extreme Programming (XP)

Extreme Programming (XP) is an "agile" software development method. The main elements of this method are described in [Bec04]. Although XP as an approach was defined and refined in, for example, software development projects at a Swiss bank, its name already indicates a strong influence by the software development culture of North America, which is defined by pragmatism. Despite its given name, XP is no hacker technology; rather, it has some very detailed, elaborate methodological aspects which have to be applied rigorously. These aspects allow its supporters to postulate that XP can be used to create high-quality software with relatively low effort, within budget, and to the satisfaction of the customers. Statistically meaningful studies of XP projects are more than mere anecdotes [DD08, RS02, RS01].

XP is rather popular in practice. Many books have already been written about XP [Bec04, JAH00, BF00, LRW02] discussing different aspects of XP in detail, as well as the current levels of knowledge about this methodology or illustrate case studies of projects that have already been conducted [NM01]. [Wak02] and [AM01] contain particular practical aids for implementing XP, [Woy08] looks at cultural aspects, and [BF00] discusses planning in XP projects.

[EH00a] and [EH01] offer a critical description of XP with an explicit discussion of its weaknesses. Amongst other things, these works criticize the lack of use of modeling techniques such as UML and draw a critical comparison with Catalysis [DW98]. A dialectic discussion of the advantages and disadvantages of Extreme Programming can be found in [KW02]. It highlights, for example, the necessity of a disciplined approach, as well as the strong and, compared to classic approaches, significantly modified demands placed on the team leader and the coach in particular.

Important elements of XP are presented and discussed below in accordance with the introductions given in [Bec04] and [Rum01]. Further topics in literature now treat XP in parallel with other methods [Han10, Leh07, Ste10, HRS09] and thus support a portfolio of agile techniques. Alternatively, they adapt agile methods for distributed teams [Eck09] or cover the migration of companies to agile methods [Eck11].

Overview of XP

XP is a lightweight method of software development. It dispenses with the need for a number of elements from classic software development in order to allow faster and more efficient coding. The potential deficits this causes for quality management are compensated for by a stronger weighting for other concepts (in particular, the test process). XP consists of a larger number of concepts. Within the scope of this overview, we will cover only the most important of these.

XP tries explicitly not to use new methodological concepts or methodological concepts that have not yet been tested to any great extent. Instead, it integrates proven techniques into a process model which focuses on the essentials and dispenses with the need for organizational ballast as far as possible. Since the ultimate goal of software development is source code, this is what XP focuses on from the very beginning. Any additional documentation is considered to be ballast that should be avoided. Creating documentation takes a lot of effort, and the documentation is often much more erroneous than the code itself because it cannot usually be analyzed and tested automatically to a satisfactory extent. In practice, customers frequently present new or modified requirements; documentation reduces the flexibility of the evolution and adaptation of the system as a quick response to these new or modified requirements. Therefore, almost no documentation is created in XP projects (with the exception of the code and the tests). To compensate for this, good comments in the source code based on coding standards and an extensive test suite are very important.

The primary goal of XP is the efficient development of high-quality software on time and within budget. The mechanisms used to do this are illustrated by the *values*, the *principles*, the basic *activities*, and the *development practices* implemented in the activities shown in the pyramid in Fig. 2.2.

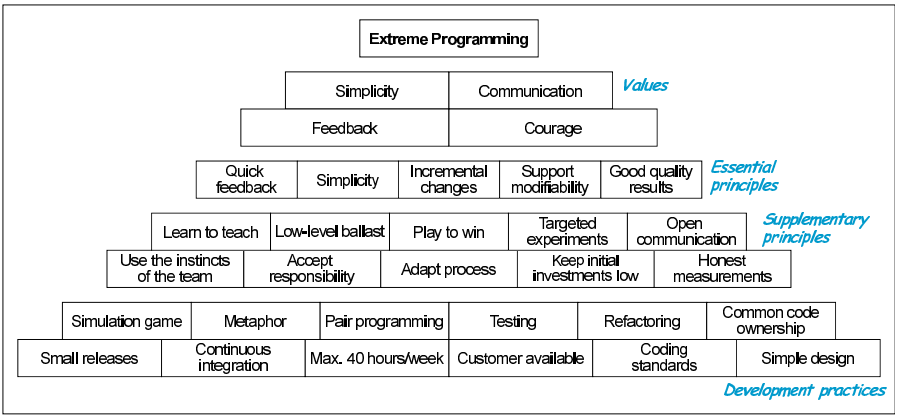

Fig. 2.2. Structure of Extreme Programming

Success Factors of XP

Now that XP has been in use for some years, we can clearly identify some of the major factors for the success of XP projects:

- The team is motivated and the working environment is suitable for XP. This means, for example, that working places are set up for pair program-

ming and the developers sit in close proximity to one another and to the customer.
- The customer is actively involved in the project and is available to answer questions. The study [RS02] showed that this has to be rated as one of the most critical success factors for XP projects.
- The importance of tests at all levels becomes clear as soon as there are any changes, new developers join the team, or the system reaches a certain size which means that manual testing can no longer be performed.
- The result of the drive for simplicity, which is a topic that is being discussed in all domains, means that documentation is omitted. Equally, the design is as simple as possible. These factors allow a significant reduction in the workload.
- The lack of system specifications and the presence of a customer who can be included in negotiations about functionality during the course of the project means that the customer is integrated in the project more intensively. This has two effects: on the one hand, it allows a fast response to changing customer wishes. On the other hand, it also allows the project to influence customer wishes. The project success thus also becomes a social agreement between the customer and the team of developers and not solely an objectively tested achievement of objectives based on documents.

This last aspect in particular corresponds to the XP philosophy of less control and a greater demand for individual responsibility and commitment. In a world in which requirements are constantly changing, this could lead to a more satisfactory result for everyone involved in the project than is possible with fixed system specifications.

Limits to the Applicability of XP

As far as project documentation and the integration of customers are concerned, XP is indeed revolutionary. Accordingly, there are a number of constraints and requirements that apply to the project size and project environment. These are discussed in various works, including [Bec04], [TFR02], and [Boe02]. XP is particularly suitable for projects with up to ten team members [Bec04] but it is evidently a problem to scale XP for large projects, as discussed in [JR01], for example. XP is simply one more approach in the Software Engineering portfolio—just like many other techniques, it can only be used under certain premises.

The basic assumptions, techniques, and concepts of XP polarize opinions. On the one hand, some programmers believe that XP elevates hacking to the status of an approach; on the other hand, XP is not taken entirely seriously because it ignores a lot of development processes that have been compiled over earlier decades. In fact, both arguments are only correct to a limited extent. On the one hand, it is true that hackers are more attracted to an XP-type

approach than to an approach according to RUP. On the other hand, when they look more closely, many software developers will recognize development practices that are already established. Furthermore, the XP approach is very strict and requires discipline for implementation.

It is certainly correct that XP is a lightweight software development method which is positioned explicitly as a counterweight to heavyweight methods such as RUP [Kru03] or V-Modell XT [HH08]. Significant differences in XP include firstly, the fact that it concentrates solely on code as a result, and secondly, the integration of the needs of the project participants. However, the most interesting difference is the increased capability of XP to respond to changes in the project environment or the user requirements flexibly. This is why XP belongs to the group of "agile methods".

The Costs of Fixing Bugs in XP Projects

One of the fundamental assumptions in XP questions important findings in Software Engineering. Previously, the assumption was that the costs for fixing errors or for implementing modifications increase exponentially over time, as described in [Boe81]. However, the assumption for XP is that these costs flatten out over the course of the project. Fig. 2.3 shows these two cost curves.

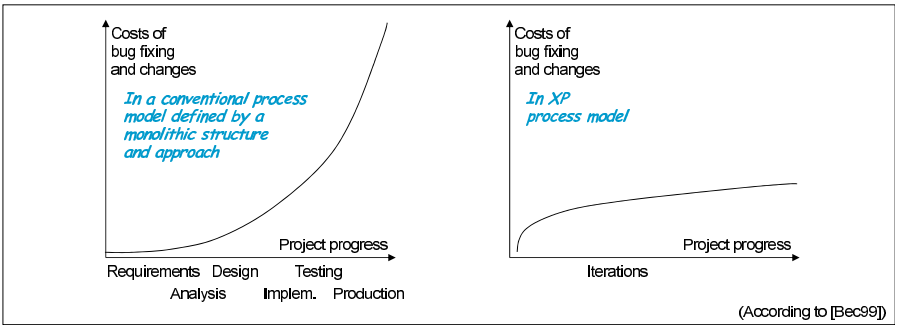

Fig. 2.3. Bug-fixing costs over the course of a project

In XP, the assumption is that the costs of modifications no longer increase dramatically over time [Bec04, Chapter 5]. There is no real empirical evidence for this assumption; however, it does have significant implications for the applicability of XP. If we assume that the cost curve can be flattened out with XP, then it is actually no longer essential to develop an initial architecture which is largely correct and which can be extended for all future developments. The entire profitability of the XP approach is therefore based on this assumption.

However, there are indicators that support at least a certain amount of flattening out of the cost curve in XP. Defects can be eliminated more quickly and more extensively by considering the following aspects, all of which are reflected in coding standards: using better languages such as Java, using better web and database technologies, and improving development practices. The use of better tools and development environments also helps in eliminating defects more effectively. Due to the common code ownership, even defects that are not localized in one artefact can be eliminated without the need for a series of planning and discussion meetings. The waiving of documentation removes the necessity of keeping any documents created consistent. On the contrary, with XP we have the effort and expense of updating automated tests. However, the fact that the tests are automated offers the significant advantage that tests which are no longer correct can be recognized efficiently—compared to the expensive and time-consuming proofreading of written documentation.

One of the main indications of a reduced cost curve, however, is an approach that uses small iterations. Errors and defects that can be localized in one iteration only have a local effect and remain within the iteration. In the subsequent iterations the effect is limited, meaning that only a slow increase in bug-fixing costs can then be expected there. The iterative approach, possibly coupled with a decomposition of the system into subsystems, may therefore produce the cost curve presented in Fig. 2.4. This is the cost curve that we saw in the auction project, for example.[1]

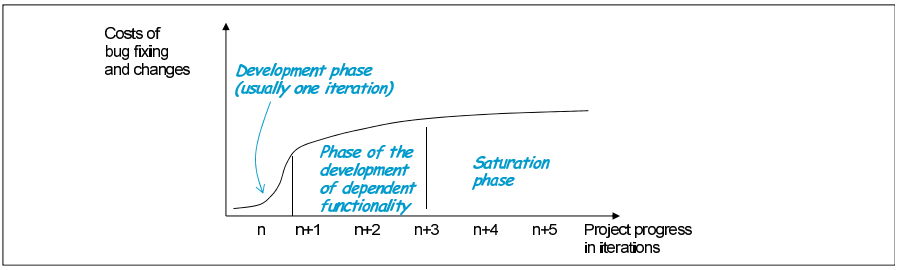

Fig. 2.4. Bug-fixing costs in iterative projects

Depending on whether the error can be localized within one part of the system, a certain increase in bug-fixing costs can arise in the causal and immediately subsequent iterations. In later iterations, it is only the costs of identifying the defect and its source that should increase. For defects in the architecture, however, which by their nature affect many parts of the system, the saturation occurs very late and at a high level. It is therefore generally worth investing a certain amount of initial expense in modeling the architecture.

[1] However, there is no statistically valid numerical data for this.

Hence, even though certain arguments support the achievable cost reduction at least to some extent, this statement must first be proven with numerical data obtained by examining a sufficient number of XP projects.

However, we can say that one of the advantages of XP is that, as a result of the development of automated tests, continuous integration, pair programming, short iteration cycles, and permanent feedback with the customer, the probability of finding errors at an early stage has increased.

Relationship between XP and CMM

In the article [Pau01], one of the authors of the Capability Maturity Model (CMM) asserts that, rather than being incompatible contradictions, XP and the software CMM [PWC+95] actually complement each other. Accordingly, XP has good development practices that satisfy core requirements of CMM. For many of the "key process areas" (KPA) demanded by CMM for the five CMM levels, XP offers practices which, although to a certain extent unconventional, are well-suited to the project area covered by the XP approach. These practices address the goals of CMM. The article [Pau01] breaks down the support provided by XP for the KPAs individually in a table. Of the total number of eighteen KPAs, seven are rated as negative and eleven are rated as positive to very positive. Of the KPAs rated as negative, however, the training program can also be rated as more positive due to the pair programming performed by experts with beginners, and the management of subcontracts can be ignored. In agreement, [Gla01] describes how the XP approach adheres to CMM Level 2 without any additional effort and indicates that the project-based part of CMM Level 3 can also be achieved with little additional effort. However, full CMM Level 3 requires cross-project measures across the company which are not addressed by XP.

In summary, [Pau01] comes to the understandable conclusion that XP has good techniques that companies could consider, but that very critical systems should not be realized exclusively with XP techniques.

Findings from Experiences with XP

XP provides a lightweight process comprised of coherent techniques. As XP focuses on the code, the process can be designed much more efficiently than RUP, for example. This efficiency means that fewer resources are required and the process is more flexible. The increased flexibility mitigates one of the basic problems of software development, namely handling user requirements that change over the duration of the project.

Ensuring the quality of the product being developed calls for pair programming and rigorous automated tests without, however, relying to much on the test theory which has been around for a long time. The constant demand for simplicity increases the quality and efficiency further.

Based on this analysis of XP, which stems partly from literature and partly from own project experiences, including the auction project described in Appendix D, Volume 1, we can categorize XP as an approach suitable for small projects with approximately 3–15 people. To a limited extent, using suitable measures, such as the hierarchical decomposition of larger projects based on components [JR01, Hes01] and the accompanying additional activities, XP can be scaled to larger tasks.

Alternatively, it may also be possible to reduce the size of the task using innovative technologies, including reusing and adapting an existing system where this is possible.

In particular, reducing the size of the task involves the improvement of used languages, tools, and class libraries. Many parts of a program that have technical code, such as the output, storage, or communication of data, have similar structures. If these program parts can be generated and, using an abstract representation of the application, composed to form executable code, this increases the efficiency of the developers even further. This is true for the product code but even more so for the development of tests which, in abstract modeling, are also made more understandable with diagrams.

Accordingly, the goal of using an executable sublanguage of UML as a high-level programming language must be to increase the efficiency of the process of developing models and transforming them into code, thereby accelerating the software development process further. Ideally, the design and implementation part of a project is reduced to such an extent that a project consists primarily of eliciting requirements that can be implemented efficiently and directly.

2.3 Selected Development Practices

Three of the development practices of XP are being investigated further: pair programming, the test-first approach, and the evolution of code. This is because they also play a significant role in agile modeling with UML.

2.3.1 Pair Programming

Pair programming existed before XP and was described in [Con95], for example. Although it was initially a stand-alone technique, today it is integrated in XP (as well as other techniques) because it is a good basis for common code ownership. It means that for all parts of the system, there are at least two people who are familiar with it.

The main idea behind pair programming is also referred to as the "principle of dual control": two developers work on one task which they solve together. They need only one computer to do so, and while one developer

types in what they have developed, the partner performs a constructive review at the same time. However, the keyboard and the control over the constructive work quickly alternate between the two parties involved. Originally, this principle was intended for use by developers with the same level of expertise; however, it is also suitable for a combination of a system expert and a project beginner. It allows the beginners to familiarize themselves with existing software structures and new techniques efficiently. In purely mathematical terms, however, pair programming initially means double the personnel expense.

In tests conducted at universities, pair programming has been studied in more detail and analysis schemes have been created [SSSH01]. The studies [WKCJ00] and [CW01] show that, for pair programming, after a relatively short time for familiarization with the new programming style, the total expense compared to programming by individuals had increased but there was a significant reduction in the duration of the project and in particular, a significant increase in the quality of the software.[2] However, it is also evident that the technique of programming in pairs has to be learned and that pair programming does not suit everyone.

Therefore, in practice rigorously enforced pair programming would not be productive. Instead, cooperatively encouraging pair programming should lead to optimal results, because among others a project also involves activities that do not need to be performed in pairs. These activities include planning activities, tool installation and maintenance, as well as (in some circumstances) discussions with customers to elicit requirements. Unfortunately, flexible working hours for developers and unfavorable room allocations are further obstacles to applying pair programming consistently.

2.3.2 Test-First Approach

Tests are performed at different points in time and discussed and used with differing intensity in different methodologies. For example: the V-Modell XT explicitly separates tests for methods, classes, subsystems, and the overall system. In the Rational Unified Process according to [Kru03], however, there is no differentiation between the test levels and no discussion of metrics for test coverage. In XP, testing is one of the four core activities—it plays a significant role and is therefore discussed in more detail here.

[Bec01] describes the advantages of the test-first approach propagated in XP for software development very clearly. [LF02] elaborates on this approach for describing unit tests and discusses the advantages and disadvantages as well as the methodological use in a pragmatic form. [PP02] compares the test-first approach with the traditional creation of tests after implementation in a

[2] Statistically meaningful example figures are given in [WKCJ00] and [CW01], showing that when pair programming was used, development costs rose by 15% but the resulting code had 15% fewer defects. This allows a reduction in costs for bug fixing. The total cost saving is estimated at between 15% and 60%.

general context. [Wak02, p. 8] contains a description of a micro development cycle based on tests and coding.

The main idea of the test-first approach is to think about test cases before developing the actual functionality (in particular, individual methods). These test cases must be suitable for checking that the functionality to be realized is correct in order to describe this functionality in the form of an example. The test cases are documented in tests that run automatically. According to [Bec01] and [LF02], this has a number of advantages:

- Defining test cases before the actual implementation allows an explicit demarcation of the functionality to be realized, meaning that the test design equates de facto to a specification.
- The test justifies the necessity of the code and describes, amongst other things, which parameters are required for the function that is to be realized. This means that the code is designed such that it *can be tested*.
- Once the functionality has been realized, the existing test cases can be used for immediate verification; this increases the confidence in the code developed enormously. Although there is no guarantee that the functionality is free of errors, practical application, including in the auction project, shows that this confidence is justified.
- It is easier to separate the logical design from the implementation. When test cases are defined—in this case before the implementation—the first step is to determine the signature of the new functionality. This signature contains the name, parameters, and the classes, which contain methods. The methods are not implemented until the next step, that is, after the definition of the tests.
- The amount of work involved in formulating test cases should generally not be forgotten, especially if complex test data is required. According to [Bec01], this leads to functions being defined in such a way that they are provided with only the data necessary in each case. This results in classes being decoupled, making the designs better and more simple.
  This argument may be true in individual cases but it may not always be correct. It would be more correct to state that classes are decoupled as a result of the early detection of the possibility for decoupling or due to retrospective refactoring. This is also demonstrated by typical examples of the test-first approach [LF02, Bec01].
- Early definition of sets of test data and signatures is also helpful in pair programming as it allows developers to discuss the desired functionality more explicitly.

One advantage of test cases is that other developers can recognize the desired functionality based on the test case descriptions. This is e.g. necessary if the code is unclear and does not have sufficient comments and there is no explicit specification of the functionality. The test cases themselves therefore represent a model for the system.

Experience shows, however, that the possibility of developing the functionality based on the tests is limited. This is because tests are usually defined less thoroughly than the actual code. Also, tests written in a programming language do not represent the actual test data very compactly or clearly.

Although the test-first approach offers a number of advantages, in practice we must still assume that defining tests in advance does not provide sufficient coverage for implementation. However, the coverage metrics that we know from the testing theory are not explicit part of XP. A very informal concept for test coverage based in particular on the intuition of the developers is generally seems satisfactory. On the one hand, this is somehow unsatisfactory for the controlling in a project, but on the other hand, it is successful in practice. But there are also tools which automatically measure the extent to which the tests detect local modifications in the code and therefore satisfy certain coverage criteria. Some are even developed or adapted for the XP approach [Moo01]. These tools include mutation tests [Voa95, KCM00, Moo01] that through simple mutation of the test object check whether a test detects the modification (defect).

If the desired functionality is implemented, once the initial test suite has been completed, the development of further tests should achieve a better coverage. These further tests include, for example, the treatment of borderline cases and the investigation of conditional expressions and loops that can be necessary in part due to the technique or framework used and therefore were not anticipated in the test cases developed in advance.

The test-first approach is generally seen as an activity which combines analysis, design, and implementation in "microcycles". However, [Bec01] also refers to the fact that test methods which are used to define test cases systematically and which measure the coverage of the code according to different criteria are generally ignored. This is consciously accepted with the argument that these test methods involve a lot more effort but the results are not (if at all) significantly better. [LF02, p. 59] at least points out that tests are created not only before implementation but also after completion of a task in order to achieve "sufficient" coverage, but does not explain precisely when the coverage is sufficient.

Depending on the type and size of the project, the strict test-first approach can be an interesting element of the software development process which can typically be used after at least an initial architecture has been modeled for the system and the system has been broken down into subsystems. By using the executable sublanguage UML/P, this approach can be elevated to the modeling level more or less unchanged. For this purpose, in a first step, sample data and sample sequences can be modeled as test cases using object diagrams and sequence diagrams respectively. Based on these test cases, the functionality to be realized can be modeled using Statecharts and class diagrams.

2.3.3 Refactoring

The desire for techniques which incrementally improve and modify source code using sets of rules is only a slightly more recent phenomenon than the creation of the first programming languages [BBB+85, Dij76]. The goal of transformational software development is to break the software development process down into small, systematically applicable steps which are manageable due to their effects being limited to the local environment. Refactoring was first discussed in [Opd92] for class diagrams. [Fow99] is highly recommended. It describes an extensive collection of transformation techniques for the programming language Java. The refactoring techniques allow us to migrate code in line with a class hierarchy. They also allow us to break down or divide classes, shift attributes, expand or outsource parts of code into separate methods, and much more. The strength of the refactoring techniques is based on how easy it is to manage the individual transformation steps (referred to as "mechanisms") and the ability to combine them flexibly, which leads to large, goal-oriented improvements in the software structure.

The goal of refactoring is to transform an existing program but preserve the semantics. Refactoring is used to improve the quality of the design whilst retaining the functionality rather than to extend the functionality. It therefore supplements the normal programming activity.

Refactoring and the evolution of functionality are complementary activities that can quickly alternate in the development process. The course of a project can thus be outlined as shown in Fig. 2.5. However, there is no objective criterion for measuring the "quality of the design". Initial approaches, for example, measure the conformance to coding standards, but are insufficient for evaluating the maintainability and testability of the architecture and implementation.

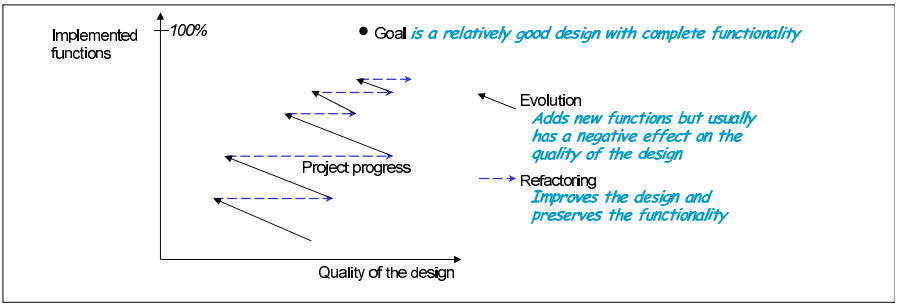

Fig. 2.5. Refactoring and evolution are complementary

No verification techniques are used to ensure that transformations which preserve semantics are correct; instead, the existing test suite is used. If we can assume that the existing test suite has a high level of quality, there is

a high probability that faulty modifications, that is, modifications which change the behavior of the system, will be detected. Refactoring often modifies internal signatures of a subsystem if, for example, a parameter is added to a method. Therefore, certain tests have to be adapted together with the code. In the sense of the test-first approach, [Pip02] even proposes refactoring first the tests and then the code.

Refactoring techniques are aimed at various levels of the system. Some refactoring rules have a small effect; others are suitable for modifying an entire system architecture. The possibility of improving a system architecture that has already been realized in code means that it has become less necessary to define a correct and stable system architecture a priori. Modifications to the system architecture of course involve high costs. In XP, however, the assumption is that maintaining system functions which are not used is more cost-intensive over the long term. In accordance with the principle of *simplicity*, the XP approach prefers to keep the system architecture simple and to only modify or extend it as required using suitable refactoring steps.

While many refactoring techniques for Java based on [Fow99] are currently under ongoing development, at present only a few similar techniques exist for UML diagrams [SPTJ01].

2.4 Agile UML-Based Approach

Due to the diversity of software development projects in terms of size, application domain, criticality, context etc., we can conclude that a standardized approach does not and will exist in the diversified project landscape.

Instead, as proposed in Crystal [Coc06], a suitable approach must be selected (and adapted) where necessary from a collection of approaches based on the following criteria: the size and type of project, the criticality of the application, as well as the experience and knowledge of the people involved in the project.

Corresponding to this diversity, this book outlines a proposal for a lightweight, agile method using UML. This proposal concentrates in particular on the technical part of an approach and does not claim to be suitable for all types of projects.

Definition of "Agility"

The characterization of the agility of a method is outlined in Fig. 2.6.

Efficiency can be improved by cleverly omitting unnecessary work (documentation, functionality not required, etc.) and also through an efficient implementation.

It is not necessarily the case that all of the techniques of an agile method focus on the *quality* of the product developed. However, certain elements of an agile method, such as concentrating on the simplicity of the design and

An approach is deemed to be "agile" if it emphasizes the following criteria:

- The *efficiency* of the overall process and the individual steps is as ideal as possible.
- The *reactivity*, that is, the speed at which the approach reacts to changing requirements or a different project environment, is high. Planning therefore tends to be *short-term and adaptable*.
- The approach itself can also be *adapted flexibly* so that it can adapt itself dynamically to internal project circumstances, as well as to external project circumstances which are determined by the environment and which therefore can only be partially controlled.
- *Simplicity* and practical implementation of the development approach and its techniques lead to simple design and implementation.
- The approach is *customer-oriented* and demands active integration of the customer during the project.
- The *capabilities, knowledge, and needs of the project participants* are accounted for in the project.

Fig. 2.6. Terminology: agility of a method

the extensive development of automated tests, do support the improvement in the quality. Other practices enforced by some agile methods, such as the lack of detailed specifications and reviews, discourage using the method for highly critical systems. In this case, an alternative process or a suitable extension must be selected.

Improved Support for Agility

The agility of a project can be improved not only by modifying the activities but also in particular by increasing the *efficiency* of the developers. It is of interest to improve the efficiency of creating models, the implementation, and the tests. This is particularly effective if the implementation and the tests can be derived as efficiently as possible or even completely automatically from models. This type of automatic generation is interesting if the source language used for the models allows a more compact representation than the implementation itself could provide.

The premise for using models in this way is that they can be created more quickly because they are more compact but still allow a complete description of the system. In the form offered in the UML standard [OMG10], UML is not sufficient for this purpose. Therefore, UML/P has been extended to include a complete programming language.

Use Cases for Models

Models that are used to generate code require a high level of detail. But then the usual coding activity, which is very time-consuming, is no longer necessary. The detailed modeling and the implementation merge into one single

activity. Code generation is therefore an important tool for using models successfully. This allows us to achieve a goal similar to XP, in which design and modeling activities are generally executed directly in code.

However, we can use models for other goals besides code generation. *Abstract* or relatively *informal* models suffice for the communication between developers. *Abstraction* means that details that are not necessary for describing the communicated content can be omitted. *Informality* means that the language correctness of the model represented does not have to be adhered to precisely. For example, diagrams on paper can use intuitive notational elements that do not belong to the language if the developers have a common understanding of their meaning as far as necessary. Furthermore, where *informal* diagrams are used, they do not have to be completely consistent with the content modeled or with other diagrams.

We can also use models to document the system. Documentation is a written form of communication intended for long-term use. A higher level of detail, greater formality, and/or consistency are therefore useful depending on the goal of the documentation. A short document that gives an overview of the architecture of a system and an introduction to important decisions that led to this architecture will have a low level of detail, but the formality and consistency within the model and with the implementation are important. If complete documentation is required, the ideal way to ensure that the documentation is consistent with the system implemented is to generate the implementation and, as far as possible, tests, from the models of the documentation.

In a project that uses UML/P for modeling and for implementation, we at least differentiate between:

- *Implementation models*
- *Test models*
- *Models for documentation*
- *Models for communication*

We can use UML/P for all of these purposes even though the models each have different characteristics. However, there is a significant advantage in the fact that there are no notational breaches between specification, implementation, and test cases. UML/P can be used for detailed as well as abstract and incomplete modeling.

Modifying Models

If a model is used exclusively for communication, then it normally exists only as an informal drawing. A model created by a tool has a formal representation of the syntax[3] and can therefore be used for further processing

[3] For example, *XML Metadata Interchange Format* (XMI) is a standard for representing UML models and therefore this type of formal representation of the syntax.

supported by tools. The syntax of UML/P is therefore precisely defined in Appendices A, Volume 1, B, Volume 1, and C, Volume 1.

We have already identified *code generation* as one of the important techniques for using models. There are two main variants of code generation: product code generation and test code generation.

The necessity of adapting software based on changing requirements also means that techniques must be available for *transforming* or *refactoring* models. Systematically adapting models to new and changed functionalities in a process validated at each stage by automated tests and invariants allows this dynamic adjustment which is already familiar from XP. As a model created according to given requirements is more compact and therefore easier to understand than the code, this also improves adaptability of models. For example, we can adapt the structure within one class diagram, which is spread across a number of files in the code. The necessity of developing a fixed architecture at an early stage of a project decreases very similar to the observations made in the XP approach. At the same time, the flexibility for integrating new functionality and therefore the agility of the project continues to increase.

We can therefore identify the following important techniques in handling models [Rum02]:

- Generating product code
- Generating test code
- Refactoring models

Furthermore, various techniques for analyzing models are interesting, such as the reachability of states in Statecharts. Another factor that is helpful in creating tests efficiently is deriving many test cases from universal specifications, such as OCL constraints or Statecharts.

Special techniques, for example, for generating database tables, generating a simple graphical interface from data models, or for migrating data sets conforming to the old model into a new model are also important and must be supported by a code generator with suitable parameters. However, this book does not cover such techniques in any further detail.

Further Useful Principles and Practices

To complete an approach, further principles and practices have to be selected on a project-specific basis. For small projects, for example, we can identify the following principles and practices, many of them adaptable from XP:

- Many small iterations and releases
- Simplicity
- Quick feedback
- Permanent dialog with a customer who is constantly available
- Short internal, daily meetings for coordination
- Review after each iteration for the purpose of process optimization

- Development of tests before implementation ("test-first")
- Pair programming as a technique for learning and review
- Common ownership for models
- Continuous integration
- Modeling standards as a requirement for the form and commenting of the models similar to coding standards

We can very easily apply the development of tests before an implementation in accordance with the test-first approach discussed in Section 2.3.2 to UML/P. To do so, we initially model sequence diagrams, which are an essential component of tests, as descriptions for use cases and we can then transform them into test cases with an acceptable level of effort. A similar situation applies for object diagrams, which we can create as examples of data sets during the recording of requirements.

Standard modeling guidelines applicable across the project are necessary to ensure the usability of the models; the project participants must be able to adapt and extend these models as necessary.

Sometimes developers still need to learn the language UML/P for modeling to become productive. Any necessary learning phase can be supported by pair programming, which will now generally be referred to as "pair modeling" in this book.

Quality, Resources, and Project Goals

The effects of the approach outlined in this section can be explained using the value system discussed in Section 2.2 and the four criteria (time consumption, costs, quality, and goal orientation) also used in XP, for example.

Communication is easier to accomplish based on UML/P models than based on implemented code if we can assume that the communication partners are familiar with UML.

Simplicity of the software is better supported for two reasons: on the one hand, as the software can be changed easily, it is even less important to establish structures for future functionality that might potentially be required and thus integrate potentially unused complexity at an early stage. On the other hand, it is even easier to remove elements that are no longer necessary from the model using refactoring steps. However, the developers themselves must still be able to recognize unnecessary complexity and superfluous functionality.[4]

Feedback is ensured by the greater efficiency in development and the resulting even shorter iterations.

Individual responsibility and courage can and must remain with the developers, as in XP.

[4] Analysis tools can only provide limited support here by recognizing which methods, parameters, or classes are not required.

The four main variables of project management are also addressed:

Time consumption: Time savings and, in particular, early availability of first usable versions ("time to market") are essential not only for Internet applications. The increased development efficiency and the higher level of reusability of technical parts allow a reduction of the time necessary.

Costs: Costs also decline due to increased efficiency and the resulting decrease in time and personnel expense. This reduction in the personnel expense means that some management efforts can also be omitted, meaning that the process used is more lightweight and thus enables further savings.

Quality: The quality of the product is influenced positively by the following factors: the more compact and therefore clearer representation of the system; the easier modeling of test cases; and through the fact that there is no longer a breach between the modeling and implementation language. Concentrating on further aspects, such as the level of test coverage, additional model reviews, and actively integrating the customer determines whether the quality of the resulting system meets the demands.

Goal orientation: In order to ensure that the system is implemented in the form desired by the user, it is important to integrate the user actively. Using UML/P does not influence this aspect in any direction, as we can assume that it is unlikely that a user would be presented with UML diagrams for discussion.

The Problems of Agile, UML-Based Software Development

In addition to the advantages discussed above, the approach outlined has some disadvantages that should be considered:

- The advantage that almost the same notation can be used for the abstract modeling and implementation can prove to be a blowback. In practice, previous approaches for an abstract modeling of essential properties, such as SDL [IT07b, IT07a] or algebraic specifications [EM85, BFG+93], have been used often as high-level programming languages as soon as their executability has been available through a tool. The same is now proposed for a sublanguage of UML.[5] If UML is used for specification, this can lead to unnecessary details being filled out for the specification because a later use of the specification as an implementation has been anticipated to early. This phenomenon is also referred to as "overengineering".
- The teaching effort for using UML/P must be assumed as high. With regard to syntax, UML/P is significantly more complex than Java, meaning

[5] In algebraic specifications, this has led at least in part to a greater focus on transformability into efficient code rather than on the abstract modeling of properties and thus led to strange implementation-oriented specifications.

that an incremental learning approach is recommended here. As an initial step, the use of UML/P for describing structures with class and object diagrams can be taught. This can be followed by sequence diagrams for modeling test cases and, building on that, OCL for defining conditions, invariants, and method specifications. Using Statecharts for modeling behavior usually requires the most practice.
However, similarly to Java, it is not only the syntax that needs to be mastered but in particular the use of modeling standards and design patterns, which must also be learned.

- At present, there is not enough tool support that covers all aspects of the desired code generation completely. In particular, efficient code generation is essential for creating the system quickly, which in turn enables efficient execution of tests and the resulting feedback.
  However, a number of tool developers are working on realizing this vision of complete tool support and are already in a position to demonstrate results.

2.5 Summary

Agile methods represent a relatively new approach which, through several characteristics, distinguishes itself explicitly from methods previously used in software development. The tools, techniques, and the understanding for the problems of software development have improved significantly. Therefore, these methods can offer more efficient and more flexible approaches for a subdomain of software development projects through, for example, strengthening the focus on the primary result, the executable system, and a reduction in the secondary activities. At the same time, the motivation and the commitment of the project participants for rigorous execution of their activities come to the fore and short iterations enable flexible, situation-dependent control of the software development.

We have identified important factors for determining the size of a project: the *size of the problem* and the *efficiency of the developers*. We can use these factors to derive the required *method size*, which consists mainly of the methodological elements to be executed, the formality required for the documents, and the additional effort for communication and management. Due to additional communication and management overheads, the *efficiency of the developers* is disproportionate in the determination of the *project size*, i.e., the number of developers required and the runtime of the project. Therefore, improving developer efficiency is a significant lever for reducing development costs.

As a modeling language for *architecture modeling*, *design*, and *implementation*, UML/P provides a standardized language framework that allows us to model the executable system and the tests completely. The compact size of the representation leads to greater efficiency and results in scaling effects for

the project size. The standardized framework prevents an otherwise often observed notational breach between the modeling, implementation, and test languages.

http://www.springer.com/978-3-319-58861-2

Agile Modeling with UML
Code Generation, Testing, Refactoring
Rumpe, B.
2017, XIII, 388 p. 176 illus., 101 illus. in color.,
Hardcover
ISBN: 978-3-319-58861-2