Orit Hazzan and Yael Dubinsky

Agile Software Engineering

Springer

Orit Hazzan, BSc, MSc, PhD, MBA
Department of Education in Technology and Science
Technion – Israel Institute of Technology
Haifa, Israel

Yael Dubinsky, BSc, MSc, PhD
Department of Computer Science
Technion – Israel Institute of Technology
Haifa, Israel

Undergraduate Topics in Computer Science ISSN 1863-7310
ISBN: 978-1-84800-198-5 e-ISBN: 978-1-84800-199-2
DOI: 10.1007/978-1-84800-199-2

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2008933449

Printed on acid-free paper

Springer Science+Business Media
springer.com

Preface

Overview and Goals

The agile approach for software development has been applied more and more extensively since the mid nineties of the 20th century. Though there are only about ten years of accumulated experience using the agile approach, it is currently conceived as one of the mainstream approaches for software development.

This book presents a complete software engineering course from the agile angle. Our intention is to present the agile approach in a holistic and comprehensive learning environment that fits both industry and academia and inspires the spirit of agile software development.

Agile software engineering is reviewed in this book through the following three perspectives:

- The **H**uman perspective, which includes cognitive and social aspects, and refers to learning and interpersonal processes between teammates, customers, and management.
- The **O**rganizational perspective, which includes managerial and cultural aspects, and refers to software project management and control.
- The **T**echnological perspective, which includes practical and technical aspects, and refers to design, testing, and coding, as well as to integration, delivery, and maintenance of software products.

Specifically, we explain and analyze how the explicit attention that agile software development gives these perspectives and their interconnections, helps

it cope with the challenges of software projects. This multifaceted perspective on software development processes is reflected in this book, among other ways, by the chapter titles, which specify dimensions of software development projects such as quality, time, abstraction, and management, rather than specific project stages, phases, or practices.

To share with the readers this multifaceted perspective, we use the **H**uman, **O**rganizational, and **T**echnical (HOT) scale for software development approaches. For example, when we refer to teamwork or abstraction levels, we emphasize the Human perspective; when software management issues are addressed, the Organizational perspective is emphasized; similarly, when the actual performance of test-driven development is described, the Technological aspect is highlighted. When the HOT? sign appears, the readers are invited to suggest their own HOT perspective.

Agile software development values these three perspectives. Therefore, in many cases, more than one perspective is illuminated by the agile approach with respect to a specific topic. Yet even when more than one perspective is significant with respect to a specific topic, we discuss from time to time only one or two main perspective(s), and the readers are invited to complete the picture.

The book is based on the authors' comprehensive experience of teaching and implementing agile software development over the past six years. A course on agile software engineering has been shaped during these years, in an iterative process that was accompanied by an ongoing research project. This course is presented in this book. In parallel to the course creation and shaping process, the agile approach has emerged and spread, becoming one of the worldwide mainstream approaches for software project management.

Organization and Features

This textbook guides a fourteen-week/session course on software engineering from the agile perspective and can be used on a weekly basis. It is intended for all who practice, research, teach, and learn software development both in academia and industry. It discusses how agile teams live and function in software development environments, how they achieve their goals, and how they act professionally in their environments. Specifically, the themes presented in the book, such as teamwork, time, quality, learning, trust, and culture, are reviewed from human, organizational, and technological perspectives, at the individual, team, and organizational levels, and are illustrated with case studies taken from industry and academia.

The fourteen chapters of the book are organized in three iterations. This structure enables us to revisit the various subjects several times during the course,

Table 1. Book structure

Iteration	Chapter #	Topic
I	1	Introduction to Agile Software Development
	2	Teamwork
	3	Customers and Users
	4	Time
	5	Measures
	6	Quality
	7	Learning
II	8	Abstraction
	9	Trust
	10	Globalization
	11	Reflection
III	12	Change
	13	Leadership
	14	Delivery and Cyclicality

as well as to guide the development of a one-release software product. Table 1 presents the book's structure book and the topic of each chapter.

Each chapter includes a theoretical approach to a specific topic, a section that refers to the given topic in learning environments, and a variety of questions and tasks for further elaboration.

The Academic Community

This book on agile software engineering can be used by instructors, academic coaches, and students as a textbook during a fourteen-week semester, either for the commonly titled "Introduction to Software Engineering" course or the "Software Engineering Methods" course.

The course is based on two main components that progress in parallel and are closely correlated with each other. The first component is theoretical and can be used in the lecture hall or the class; the second is software project development guided by the agile approach that takes place in a physical learning environment that we call a studio or lab.

This book is written for the entire course community—students, instructors, and academic coaches. Students are the learners who become familiar with the agile approach both from a theoretical perspective (in the lectures) and from a practical perspective (in the studio). Instructors are the teachers of the course's theoretical ideas, who usually teach in a class or in a lecture hall; yet, interactive teaching and active learning can be facilitated in this setting as well. The academic coaches are the practitioners who guide the software project development

in the studio (we elaborate on this role in Chapter 1, Introduction to Agile Software Engineering).

The positive results of agile software projects, as elaborated throughout the various chapters of the book, are not the only motive for this course, which presents the field of software engineering from the agile perspective. There are three additional characteristics of the course, which are especially relevant when it is taught in academia.

First, the agile approach was developed by practitioners working in the software industry, and has become mainstream in that industry. Therefore, it makes sense to articulate its nature and main concepts to prospective software engineers in the framework of a course that deals with software engineering.

Second, teaching a software engineering course within the framework of agile software development emphasizes a comprehensive image of the field. This is because agile software development explicitly addresses human, organizational, and technological aspects of the software development process with respect to all players participating in that process. Thus, the agile approach serves as an opportunity to draw this comprehensive and complex picture of the field.

Third, according to the Software Engineering 2004 Curriculum, developed by the IEEE Computer Society and the Association for Computing Machinery Joint Task Force (see http://sites.computer.org/ccse/SE2004Volume.pdf), software engineering students should acquire additional skills beyond the technical and scientific ones. One illustrative example is teamwork-related skills. Since teamwork is one of the basic ideas of agile software development, it is only natural to integrate teamwork-oriented skills in the teaching and learning process of software engineering from the agile perspective. Furthermore, since it is natural to teach agile software development in a teamwork-oriented environment, there is no need to introduce the topic of teamwork artificially; rather, a teamwork-based learning environment can be used to teach this topic. This element is emphasized mainly, but not only, in the studio element of the course.

Suggested Uses in an Academic Environment

Each chapter presents a full week of the course: two weekly lecture hours and a four-hour weekly studio meeting. The first part of each chapter includes contents suitable to be presented in the lecture. This part usually presents material beyond what it is possible to teach in a two-hour lecture. Therefore, it is advisable not to try to deliver all the content in two hours; rather, we suggest selecting from each chapter the most relevant topics to be discussed with each particular class of students. It is also advisable to encourage in the lectures some active learning elements, as is suggested in the various chapters. The second part of each chapter

addresses the teaching and learning of the chapter topic. It presents teaching and learning principles and the activities conducted in the studio each week.

As preparation for the next week's lectures and studio meeting, instructors and academic coaches can ask the students to read the relevant chapter and to work on selected activities presented throughout the body of each chapter. The students' preparation for the lecture will also partially solve the time limitation problem of addressing all the ideas presented.

Finally, though the book presents a full fourteen-week semester course, which consists of two weekly lecture hours and four-hour weekly studio meetings, it is possible to teach only one component of the course. The material provided in this book enables each instructor/academic coach to make the needed adjustments.

The Industrial Community

Since agile development has become one of the mainstream approaches for managing software projects, more and more software organizations of different sizes and types ask themselves whether the agile approach fits them. Even when it is found that agile software development is relevant for a given organization, questions such as the following are usually asked: How can we manage a transition to the agile software development process? How can our organization cope with the changes required for such a transition? How can we teach agile software development to all the software practitioners and all the other software project's stakeholders?

This book, when used in an industrial setting, aims to answer these and other relevant questions which software organizations face when dealing with the transition to agile software development. For example, in Chapter 12, Change, we discuss how to initiate a transition process to agile software development in an organization. When the organization has already transitioned to agile software development, the book can also be used for answering questions related to the actual implementation of agile software development in the organization. For example, in Chapter 2, Teamwork, we discuss how teams can be formed to exploit their potential, to avoid conflicts, and to solve dilemmas.

Suggested Uses in an Industrial Environment

This book can be used in industrial settings by coaches of software teams, software team leaders, and facilitators of agile software development workshops, both for the teaching and learning of agile software development, as well as for its implementation. The book can also be used by interested software practitioners who are not necessarily within a formal teaching framework.

We propose two ways to use the book in industrial environments.

First, the book can be used for a course which is based on 14 sessions. This course format fits for organizations that wish to expand their members' professional knowledge by becoming familiar with agile software development, without necessarily implementing the agile approach. If the course also contains the development of a software project using the agile approach, which in academia takes place in the studio, a new software system should be developed for learning process purposes, with respect to which the different activities are facilitated. The development of a new software project should be undertaken whether the course is taught to a real team or to a group of people from different teams or organization. In the case of a real team, the development of another project than the team's real project will enable the team not to confuse their current work habits with agile practices.

Second, for organizations which wish to start implementing agile software development right away or in the near future, we suggest that the agile approach be taught first in a short format of a two-day workshop to a team that has been carefully selected to start the transition to agile software development within the organization. Chapter 12, Change, elaborates on such a transition process, explains the motivation and rationale for this intense workshop format, and outlines the workshop schedule. After the team members have participated in that workshop, and when the team starts implementing agile software development with its real project, the book can be used for clarifications and elaborations.

In both cases, as well as in other learning environments in industry, the teaching and learning principles presented in the book can naturally be applied.

Acknowledgments

We would like to thank all the practitioners, researchers, students, and mangers, both in academia and in the software industry, who during the past six years shared with us their professional knowledge, experience, thoughts, and feelings with respect to agile software development. They all contributed to our understanding of the nature of agile software engineering and fostered our shaping of the approach presented in this book.

Contents

1
Introduction to Agile Software Development

Abstract

What is agile software development? Why is an agile perspective on software engineering needed? What are the main characteristics of agile software development? What can be achieved by agile software development processes? Does agile software development form a pleasant and professional software development environment? These are the main questions addressed in this introductory chapter of the book. After reading this chapter, not only will you gain some insights about agile software development in general, you will also understand the nature of agile software development and be able to clarify how it establishes a professional software development environment in which software engineers are able to express their skills and, at the same time, to produce quality software products. In the section that deals with the learning of agile software development, we launch the development of a software project, describe the studio as the development workplace, and explain the development schedule. The ideas presented in this chapter are further elaborated in the following chapters of the book.

1.1 Overview

This chapter introduces you to the world of *agile* software development. It does so by discussing the main ideas that form the basis for the agile approach and the main characteristics of agile software development. Data about agile projects are also presented and explained.

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_1, © Springer-Verlag London Limited 2008


HOT

In general, agile software development offers a professional approach to software development that encompasses human, organizational, and technological aspects of software development processes.

Specifically, the main ideas of agile software development are first introduced by describing the Agile Manifesto and its implementations, and second by presenting specific agile practices that enable agile teams to accomplish their development task with high quality. This background enables us to explain the data we present with respect to agile processes and products.

In the part of this chapter that deals with learning agile software development, we launch the software development project that accompanies the entire book. We describe the studio as the development workplace in academia and present the development schedule.

This introductory chapter forms the basis for the understanding of the following chapters of the book.

1.2 Objectives

- Readers will become familiar with the concept of software development methods.
- Readers will become familiar with the concept of agile software engineering.
- Readers will understand the nature of the agile software development process.
- Readers will gain some practical notion of agile software engineering.
- Readers will get an overview of the subject taught in the book.

1.3 Study Questions

1. Based on your current experience in software development, specify the main problems which you face in this process. Compare your experience with the literature on software projects.
2. Describe the practices, processes, and guidelines that characterize your software development.
3. Read the Agile Manifesto at http://www.agilemanifesto.org/. What can be learned from this website about the agile software development movement? About the people who initiated it? About the process itself? About the agile software development environment?

4. When did the agile approach emerge? What was the background of its emergence and evolution?
5. What are the three main characteristics of agile software development? In what ways does agile software development differ from other software development approaches?

1.4 Three Perspectives on Software Engineering

Software engineering is the profession that applies scientific knowledge to the construction of software products needed by customers. The scientific knowledge in the case of software engineering comprises mathematics, computer science and the specific domain that the developed software deals with. In order to achieve their goals, software developers should possess professional knowledge and know how to apply it. Different approaches to software engineering exist; this book focuses on how the agile approach is applied.

One of the basic tools that developers need in order to accomplish their task is a well defined engineering process described by a software development method. A software development method is a set of activities and practices, as well as roles and norms of behavior, derived from a set of professional aims, which are carried out in a logical and specified order.

A software development method should address not only the technological aspects, but also the work environment and the professional framework. Accordingly, agile software engineering is reviewed in this book from the following three perspectives:

- The **H**uman perspective, which includes cognitive and social aspects, and refers to learning and interpersonal (teammates, customers, management) processes.
- The **O**rganizational perspective, which includes managerial and cultural aspects, and refers to the workspace and issues that extend beyond the team.
- The **T**echnological perspective, which includes practical and technical aspects, and refers to how-to and code-related issues.

Specifically, we explain how the attention that agile software development gives these aspects helps cope with the challenges of software projects. To highlight this multifaceted perspective of the agile approach, we introduce the Human, Organizational, and Technical (HOT) analysis scale for software development. See Figure 1.1 for a schematic view of the HOT analysis framework.

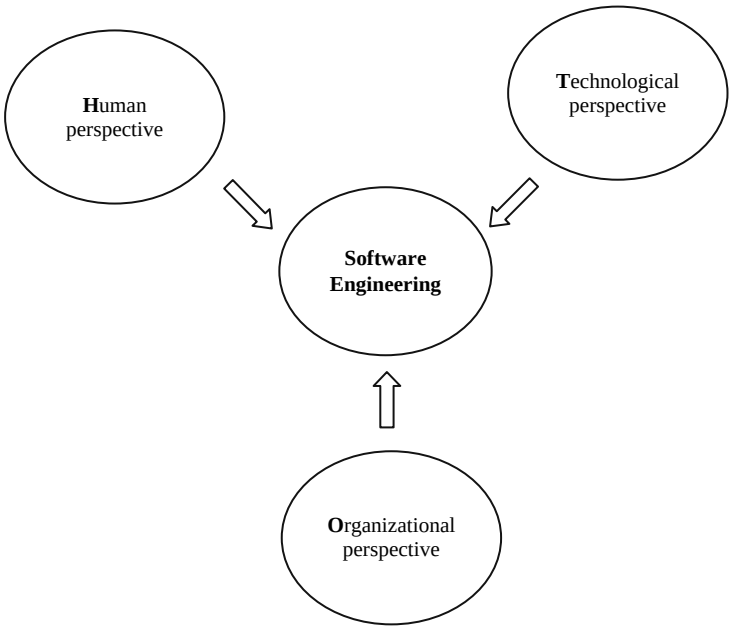


Figure 1.1 *The HOT analysis framework for software engineering.*

Tasks

1. With respect to each of the aspects mentioned above—human, organizational, and technological—mention at least two topics related to software engineering that in your opinion belong in that category.
2. Describe the development process you used for the last software project you worked on. Analyze the process benefits and pitfalls according to the HOT scale.

1.5 The Agile Manifesto

Figure 1.2 presents the Agile Manifesto. It was formulated by seventeen software developers who gathered in February 2001 in the Wasatch Mountains of Utah to try to find common ground for their perceptions of the software development process and to formulate the common elements of what some of them had already implemented in their different software organizations. The outcome of that meeting was the Agile Manifesto, which presents an alternative approach to the

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Figure 1.2 *Manifesto for agile software development.*

software development process that had been applied during the past forty years, from the early stages of the development of complex software systems.

The mere formulation of the Agile Manifesto implies that though there are agreed upon common principles and ideas, they can be applied differently by specific development methods. Indeed, the Agile Manifesto is applied by different agile methods, such as Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, and others.

In what follows we examine the Agile Manifesto.

Task

If you are familiar with at least one agile method, during the following explanation of the Agile Manifesto, analyze how it is applied by the agile method with which you are familiar.

1.5.1 Individuals and Interactions over Processes and Tools

This principle encourages us to focus on the individuals involved in the development process, rather than on the process and/or the tools. Specifically, this principle encourages software developers to give high priority to the people who participate in the development process, as well as to their interaction and communication when they develop, interact, think, discuss, and make decisions with respect to different issues related to the software development process and environment. In other words, according to this principle, one of the first considerations that should be taken into account when a decision related to the development process is made, is the influence of the decision's outcome on the people who are part of the development environment as well as on their relationships and communications.

For example, instead of investing effort in the maintenance of a development method by using state-of-the-art hard-to-use tools with difficult-to-follow procedures that result in a useless output, that effort should be channeled to the construction of a development environment that enables each of the participants (teammates, customers, management) to understand the development process, to become part of it, to contribute to it, and to collaborate with all the other project stakeholders.

1.5.2 Working Software over Comprehensive Documentation

This principle affirms that the main goal of software projects is to produce quality software products. This idea has three main implications.

First, agile software development focuses on the development itself and the creation of only those documents that are needed for the development process. Some of these essential documents, depending on their characteristics and usefulness, may be posted on the wall of the agile collaborative workspace so that they will be accessible to *all* the project stakeholders *all the time*.

Second, agile software development processes start the product's actual development (that is, coding) as soon as possible, in order to get some sense of the developed product. This early development enables both the teammates and the customers to improve their understanding of the developed product and to proceed with the development process on a safer ground.

Third, from the customers' perspective, this principle helps ensure that customers get a bugless high quality system that meets their requirements. This, of course, has direct implication for the quality-related activities that agile teams perform.

As can be seen, this principle supports the first principle of the Agile Manifesto by binding the people who participate in the development process with the actual development process. Such a connection inspires a culture in which software quality is one of the main values.

The importance of this principle is highlighted when its implications are compared with those of development processes which postpone either the development stage (sometimes for several years) or quality-related activities (mainly testing). In the first case, the fact that production starts only after a lot of documentation has been produced (which presumably, though not in practice, includes all the customer requirements), neglects the reality that software development is characterized by many changes and is based on a gradual learning process. As a result, in many cases, development that prepares in advance a lot of documentation without starting the actual development, cannot provide the customer the needed system; therefore, inconsistency exists between the project documentation and the actual product. In the second case, the postponement of quality-related activities leads to a situation in which the practitioners involved in the development process

cannot cope successfully with the complexity of testing, either from a cognitive or a managerial perspective (see also Chapter 6, Quality).

1.5.3 Customer Collaboration over Contract Negotiation


HOT

This principle changes the perception of the customer's role in software development. It encourages agile software developers to base their work on ongoing and daily contact with the customer. Such a close contact enables customers to cope successfully with the changes that characterize software projects.

Human interrelationships, mainly between customer and management, are emphasized by this principle of the manifesto. These interrelations in turn have direct implications on the development team, which should employ specific practices to ensure this kind of relationship and communication. Such practices, when employed on a daily basis, directly influence the culture of agile organizations.

This principle also points to a conceptual change with respect to the nature and formulation of software product contracts.

Thus, by dealing with contract- and communication-related issues aimed at ensuring that the customer gets the desired product, this principle of the Agile Manifesto further supports the second principle of the Agile Manifesto.

1.5.4 Responding to Change over Following a Plan


HOT

This principle encourages agile software developers to establish a process that copes successfully with changes introduced during development, without compromising the high quality of the developed product. The rationale for this principle is the recognition that customers cannot predict a priori all their requirements; therefore, a process should be established in which the requirements as gradually understood by the customer can be shared with the teammates. Accordingly, agile software development allows the introduction of changes in the developed product which have emerged from an improved understanding of the software requirements, without necessarily increasing the cost of development. This process is explained mainly in Chapter 3, Customers and Users, and Chapter 4, Time.

Tasks

1. Based on your personal experience with software development, for each principle of the Agile Manifesto suggest a situation that shows how that principle may influence actual software development processes and environments.

2. Discuss the connections among the four principles of the Agile Manifesto. Specifically, address how each principle supports and is supported by the other principles.
3. Specify how the HOT perspectives are expressed in each of the four principles of the Agile Manifesto and in the Manifesto as a whole.

1.6 Application of Agile Software Development

Based on common understandings encapsulated by the Agile Manifesto, the agile approach is applied by several development methods, each one implementing the main ideas of the agile approach according to its priorities, preferences, and perceptions of what a software development method should look like.

In this section we introduce several of the basic practices of agile software development that most of the agile methods apply to some degree. The term *practice* means a specific activity that its application supports the principle(s) of the development approach.

Task

While reading the following practice descriptions, for each of them:

- Analyze it from the HOT perspectives.
- Explain how it is derived from the Agile Manifesto.

Whole team. The practice of Whole Team means that the development team (including all developers and the customer) should communicate in a face-to-face fashion as much as possible. The practice is applied in several ways.

First, the development team is located in a collaborative workspace—a space which supports and facilitates communication. Second, all team members participate in all the product presentations to the customer, hear the customer's requirements, and are active in the actual planning process (see Chapter 3, Customers and Users). Third, participants that traditionally belong to separate teams (e.g., testers and designers), are integrated into the development team and process.

In the Whole Team concept each team member has an additional role besides that of developer (see Chapter 2, Teamwork). The rationale for this role distribution is that one person, usually the team leader, no matter how skilled he or she is, cannot handle in a professional manner all of the essential and complex responsibilities involved in software development. The distribution of responsibilities in the form of roles helps to control and manage these responsibilities. In addition,

this scheme ensures that all team members are involved in all parts of the developed software, each from his or her role perspective.

Each day, during the development hours (see Chapter 4, Time), the team is located in one space; in addition, each team member has a private space for personal tasks and professional tasks that need to be carried out individually. The walls of the development workspace serve as a communication means, constituting an informative and collaborative workspace. The information posted on the walls includes, among other relevant items, the status of the personal tasks that belong to the current iteration and the measures taken. Thus, all the project stakeholders can be updated at a glance at any time about the project's progress. In addition, the entire team holds daily stand-up meetings, which usually take place in the morning. In these meetings, each team member presents in two or three sentences the status of his or her development tasks and what he or she plans to do during the day to come, with respect to both the development tasks and the personal role.

Short releases. Agile software development processes are based on short releases of about two months, divided into short iterations of one or two weeks, during which the scope of what is to be developed in that iteration is not changed. At the end of each iteration the software is presented to the customer and the customer provides feedback to the team and sets the development requirements for the next iteration.

The detailed plan of each short iteration is carried out during a full day—a Business Day—which is specifically allocated for this purpose at the beginning of each iteration (see Chapter 3, Customers and Users). In the Business Day all the project stakeholders participate—customer, team members, users, management representatives, representatives of related projects, and so on. The Business Day includes three main parts: a presentation of what was developed in the previous iteration, along with any relevant measures taken; a short reflective session in which the development process performed so far is analyzed and lessons are learnt; and the actual planning of the next iteration. At the end of the Business Day, a balanced workload is ensured among all team members.

The nature of the activities that take place during the Business Day, and the fact that a Business Day takes place every week or two, enables all the project stakeholders to construct their knowledge related to the development product and process gradually, based on what they see, hear, and perform during each iteration. Specifically, during this process, the teammates improve their understanding of what should be developed, mainly because they hear the requirements directly from the customer during the planning session.

Time estimations. In agile software development two important practices are performed with respect to time estimation. First, the teammate who is in charge of the development of a specific task also estimates the time needed for its

development; this practice increases the team member's responsibility and commitment to the project. Second, development tasks are formulated in a way that allows their time estimation to be set in hour resolution. This fact is important because the greater a development task is, the harder it is to estimate its development time, and vise versa: the smaller the time segment estimated, the more accurate its estimation is. Consequently, the development pace can be planned more precisely. This encourages a culture in which plans can be set and followed in such a way that deadlines should not be postponed.

From the team perspective, since time estimations are performed during the Business Day with full team attendance, all teammates know what each team member has committed to in terms of development tasks and time estimations. This fact increases the project's transparency, and consequently, the teammate's responsibility to perform well. Also, the load balance that is ensured among all team members further reinforces the trust and communication among them.

Measures. In order to navigate the development process so that a high quality product is produced, agile software development processes are accompanied by various measures about which all the project stakeholders decide according to their needs. The importance attributed to these measures is expressed, among other means, by a special role—that of tracker—assigned to one team member, who is in charge of the measures.

Measures enable the team to improve the development process, and consequently, the developed software. Measures also convey the message that the development process should be monitored and that this monitoring should be transparent to all participating developers. Chapter 5, Measures, further elaborates on this practice.

Customer collaboration. Agile software development methods encourage the customer to become part of the development process. The goal is to get ongoing feedback from the customers and to proceed according to their needs. This avoids the need for speculation, which may lead to incorrect working assumptions.

This practice implies that in agile software development all team members have access to the customer during the entire development process. This direct communication channel increases both individual interactions and the chances that the software requirements are communicated correctly. Consequently, it helps the teammates to cope successfully with the system development process: first, there is no need to speculate about the customer's needs; second, the overhead of introducing change at later stages is reduced significantly. The practice of customer collaboration is elaborated in Chapter 3, Customers and Users.

Test-driven development. Test-driven software development encourages developers to build automatic unit and acceptance tests.

Unit tests are written prior to code writing in a gradual process: each step starts with writing a specific test case followed by adding a small functionality that lets the test be successful. This implies that thought must be given to the development task before actual coding starts. It also helps control the development process by clarifying what has been developed and tested so far.

Acceptance tests, which are defined by the customer and outline how each functionality should be tested, clarify the requirements for the teammates and lead to the development of a product that meets the customers' needs.

An automatic testing process makes it possible to run tests effortlessly at any stage of the development process, and to verify by regression tests that no previous development was damaged as a result of new developments.

Thus, in agile software development, both unit and acceptance tests are integrated within the development process itself, instituted by the teammate who developed the code, and are not transferred to the responsibility of the quality assurance department at a later stage. Consequently, the shift of responsibility to another department, in most cases the quality assurance department, is eliminated.

Such a process reflects a culture in which quality is highlighted; it also clarifies who is in charge of the testing of each developed unit. Chapter 6, Quality, further elaborates on this practice.

Pair programming. This practice means that each code is developed by two teammates who sit together in front of a computer and in an interactive, communication-based process work on a specific development task (Williams et al. 2000). It is important to note that even though the development is carried out in pairs, there is personal responsibility for each development task.

During pair programming it is harder to be distracted, and hence pairs tend to remain focused on the development task. In addition, each task is characterized by two levels of abstraction: that of the driver—the one who works with the keyboard and thus thinks on a lower level of abstraction, and that of the navigator—the mate who can think about the development task on a higher level of abstraction.

The application of this practice implies that all team members will become familiar with all parts of the developed software and improve their comprehension of the entire project. This fact encourages a culture that is characterized by knowledge sharing.

Refactoring. The practice of refactoring encourages teammates to improve code readability without adding functionality. The mere inclusion of refactoring in agile software development acknowledges that one cannot, a priori, predict all development details; it therefore legitimates time dedicated to improving software readability without pressure to move on to the next developmental task. The application of this practice reduces cognitive complexity. Refactoring is further discussed in Chapter 8, Abstraction.

Additional agile practices. This, of course, is not the entire story. Additional agile practices exist that support agile software development. Some of them are reviewed later in the book.

Tasks

1. For each of the above agile practices, suggest how it might have influenced a software project in which you have participated.
2. Based on the above description of the basic agile practices, fill in the following Table 1.1 by explaining how each agile practice supports each of the Agile Manifesto principles.
3. Based on the above description of the basic agile practices, fill in the following Table 1.2by explaining how each agile practice supports each of the HOT perspectives on agile software development. If possible, refer also to sub-aspects.

Table 1.1 Practices to support the Agile Manifesto

	Agile Principle I	Agile Principle II	Agile Principle III	Agile Principle IV
Whole team				
Short releases				
Time estimations				
Measures				
Customer collaboration				
Test-driven development				
Pair programming				
Refactoring				

Table 1.2 Practices according to the HOT scale

	Human: cognitive and social aspect	Organizational: managerial and cultural aspect	Technological: practical and technical aspect
Whole team			
Short releases			
Time estimations			
Measures			
Customer collaboration			
Test-driven development			
Pair programming			
Refactoring			

1.7 Data About Agile Software Development

Data indicate that agile software projects cope successfully with the common problems of software projects. For example, in the "State of Agile Development" survey[1] conducted by VersionOne and the Agile Alliance in 2007, 60% of the respondents estimated a 25% or greater improvement in time-to-market, and 86% of the respondents estimated a 10% or greater improvement in time-to-market. 55% of the respondents reported a 25% or greater improvement in productivity, and 87% of the respondents reported a 10% or greater improvement in productivity. 55% of the respondents reported a 25% or greater reduction in software defects, and 86% of the respondents reported a 10% or greater reduction in software defects.

Tasks

1. Compare these data with the data related to other software development approaches.
2. How can these data about agile software development be explained by the agile practices previously presented?

The data about the reduction in time to market are now explained based on the different agile practices discussed in the previous section.

Code is easier to work with. It is easier to work with code developed through an agile process mainly due to the *test-driven development*, *pair programming*, and *refactoring* practices. These agile practices demonstrate that since software development is a complex process, early investment in the code quality results in code that is easier to work with in later stages.

Task

Explain how *test-driven development*, *pair programming*, and *refactoring* ease code development processes.

Development is manageable and controlled. Due to the *short iterations/releases* practice, in each iteration only a small portion of the development tasks are dealt with, estimated, and developed. As has been mentioned before, this

[1] Agile Development: Results Delivered: http://www.versionone.net/pdf/AgileDevelopment_ResultsDelivered.pdf

fact eases the time estimation and development processes. Since the estimation of small software chunks is more accurate and is made in hour resolution, it is easier to observe whether the development pace is kept or whether an adjustment should be made in the project plan.

The customer's needs are met. Since the *customer* is involved in the project development for the entire process and shares with the development team his or her vision during the entire process, only what he or she articulates and needs is developed. Thus, the extra effort and time teammates sometimes invest in developing what they just suppose to be the customer's needs, without having a way to verify that this is indeed what is requested by the customer, are saved and put entirely into fulfilling and developing the customer's actual requirements.

In addition, as part of the customer's role, he or she defines *acceptance tests* which aim at verifying that what the customer wants to be developed is really developed. Thus, once again, any divergence from what the customers ask for is detected earlier in the development process and efforts that could have been dedicated for useless functionality are saved and channeled towards useful purposes.

HOT

Production increment. Since the process of software development is a hard task, and in many cases involves solving difficult problems, developers might be distracted by external factors which are not directly related to the development process. Since agile teams employ the *pair programming* practice, each teammate in the pair helps the other stay focused, and together they can cope with the challenges they encounter.

Knowledge sharing. Due to *pair programming* and *collaborative workspace*, as well other practices that agile methods employ, knowledge sharing is fostered in agile software development environments in general and among agile team members in particular. Knowledge sharing is of course very useful for the development process itself. It has benefits also when one team member leaves the team. In such a case, because of ongoing knowledge sharing, the development process can continue, without having to invest time in relearning the knowledge that the teammate who leaves has acquired over the years.

Tasks

1. What other agile practices foster knowledge sharing in agile teams?
2. Describe a scenario taken from a software project which illustrates how each of the above factors reduces the time-to-market of software products developed by agile software development teams.

1.8 Agile Software Development in Learning Environments

1.8.1 University Course Structure

This book presents a fourteen-week course on software engineering from the agile perspective. The course consists of two weekly lecture-hours and four weekly studio-hours. The lectures aim at addressing the relevant topics from a more theoretical yet active-based perspective. All the students registered for the course attend the same lecture. In the studio, students are split into teams of ten to twelve students, and each team develops a software project by an agile process, supervised by an academic coach (we elaborate on this role later in this chapter).

This course structure provides a pedagogical opportunity to develop a software product through an agile process as well as to enhance communication among the students and between the students and the lecturer. For example, when a specific topic is presented in the lecture, students can share and reflect on the different experiences gained in each studio with respect to that topic; this reflection is heard by all the students, the academic coaches, and the instructor.

Inspired by the agile approach that supports learning processes (see Chapter 7, Learning), both the lecture series and the studio series are composed of three iterations. With respect to the lectures, this course structure implies that after the students gain a fundamental understanding of some idea, new topics are presented and some topics are re-reviewed based on this understanding. In the studio, the course structure establishes an iteration-based software development process. Table 1.3 presents the course structure.

1.8.2 Teaching and Learning Principles

In the chapters of this book we present teaching and learning principles. These principles are presented as pedagogical guidelines for the teaching of *any* software development approach and can be applied in both academic and industrial settings. For each teaching and learning principle, we elaborate its general pedagogical merit and how it is implemented when the agile approach is taught. In the

Table 1.3 The course structure

Week #	Lectures	Studio meetings
1–7	Lectures: Iteration I	Studio: Iteration I
8–11	Lectures: Iteration II	Studio: Iteration II
12–14	Lectures: Iteration III	Studio: Iteration III

last chapter of the book, Chapter 14, Delivery and Cyclicality, we summarize all the principles in one table.

Additional details about these principles can be found in Hazzan and Dubinsky (2003, 2006, 2007).

In this section we present two principles.

1.8.2.1 Teaching and Learning Principle1: Inspire the Software Development Approach

This is a meta-principle that integrates several of the principles described later on and, at the same time, is supported by them. It suggests *inspiring* the software development approach that is taught instead of *lecturing* about it.

The application of this principle is expressed mainly by active learning, on which the next principle elaborates. In addition, it is reflected in the teaching environment. More specifically, active learning-based lessons should take place at a site that enables the actual performance of the software development approach. Accordingly, the studio meetings of the course, in which the students develop a software product using an agile process, take place in computer labs that reflect an agile software development environment and include a large table for the planning sessions, computers arranged for pair programming, and flipcharts or whiteboards to elicit communication processes and to establish a collaborative workspace.

1.8.2.2 Teaching and Learning Principle 2: Let the Learners Experience the Software Development Approach

This principle is derived directly from the previous one. In fact, both principles stem from the importance attributed to the learners' experimental basis, which is essential in the learning of complex concepts. This assertion is in line with the constructivist perspective on learning, whose origins are rooted in Jean Piaget's studies. According to the constructivist approach, learners construct new knowledge by rearranging and refining their existing knowledge (Davis et al. 1990; Smith et al. 1993). In this process, mental structures are developed in steps, each step elaborating on the preceding ones. Chapter 7, Learning, further elaborates on this concept.

Accordingly, since a software development approach is a complex concept, its gradual learning process should be based on the learner's experience. One way to support and to enhance such a gradual mental learning process is to adopt an active learning teaching approach, according to which learners are *active* to the

extent that enables a reflective process (see Teaching and Learning Principle 4 in Chapter 7). In addition, the course structure is iterative. This allows learners to revisit the taught concepts several times during the course and to refine their understanding of the topics comprising the software development approach.

1.8.3 The Studio Environment

The studio is a learning environment in which an intensive student-student and student-academic coach interaction leads the software development process (Kuhn 1998, Tomayko 1996, Hazzan 2002).

The studio is the basic learning method used in architecture schools and has been central to architectural training for most of the twentieth century. In the architectural studio, a group of students meet their academic coach three times each week. Such an intensive schedule puts pressure on the students and, as a result, they devote themselves to the art of design: students learn the skills, the professional language, and the discipline's ways of thinking. In addition, the studio is the place where the students reveal their personalities and can express their professional skills. Moreover, this learning environment exposes students to different kinds of social interactions, such as working on team projects, contributing to class discussions, and presenting their products in front of their classmates.

Based on the suitability of the studio approach to architecture education, it is suggested applying this teaching approach to other disciplines, mainly to professions in which design considerations are inherent, there is more than one approach to a given problem, and a solution to a given problem is usually not unique. It seems clear that software engineering fits perfectly with the adoption of the studio as a learning environment.

With respect to the context of this book, we now answer the question of how the studio concept can be applied when software development methods in general, and agile software development in particular, are taught.

First, since the studio should reflect an agile software development environment (see Teaching and Learning Principle 1, presented above), it should serve as a project development center. Accordingly, each studio should include computers for the development itself and a table in the middle of the studio for planning sessions and other activities. In order to promote pair programming, the number of computers in each studio should be equal to half the number of students in each studio plus one integration machine. Thus, students naturally begin programming in pairs. On the walls there are whiteboards that reflect the project status, presenting project stories, measures, and additional material (e.g., the roles in the team). Such a learning environment supports a teamwork-based development

process and delivers a clear and coherent message to the students about what is expected from them.

Second, as far as the desire to teach agile software engineering in an environment that resembles, as much as possible, real software development environments is concerned, a university course framework sometimes requires adjustment. This is mainly because students take other courses in parallel and cannot dedicate the entire week to their work on the development of the course software project. Thus, for example, the weekly studio meetings are supported by electronic communications.

Third, since reflection is encouraged by agile software development, reflection questions are presented to the students after each studio meeting and can be submitted through an online course management tool. In these reflections, students should be encouraged to express not only positive ideas, but also negative feelings and suggestions for improvement.

Throughout the studio meeting descriptions in this book, additional details about the nature of the studio are added. Additional details about the course structure in general and the studio environment in particular are presented in Dubinsky and Hazzan (2005).

1.8.4 The Academic Coach Role

Each team of students is guided by an academic coach who is in charge of the project management (but *not* of the project development), as well as of the student evaluation.

The academic coach supervises the development project and acts as the *academic* customer of the product as well as the *academic* coach of the team. As the academic customer, the academic coach sets requirements from the academic point of view and, if needed, changes them accordingly as the development project progresses. The academic coach also ensures the students' smooth development of their tasks and the performance of their personal roles. The academic coach interaction with the student team consists of keeping the course schedule while reflecting on the project's progress briefly in the weekly studio meetings, evaluating the students' performance as a team and as individuals, and taking care of the administrative infrastructure.

When there are several student teams, the academic coaches form a coaching team. It is recommended that the coaching team meet every week (if possible, with the course instructor) after or before the meeting with the students.

General decisions about the studio component should be made jointly by all the coaching team members as much as possible, leaving each academic coach the

freedom to adjust the particular studio he or she guides to his or her pedagogical and professional beliefs.

When a new academic coach joins the coaching team, he or she is trained by participating in all the semester sessions of one or more of the other academic coaches.

When a team of academic coaches experiences the studio environment for the first time, a transformation in their conception of the teaching and learning process is required. For example, academic coaches are physically present in most of the studio meetings, alongside the students, during the actual software development process. In addition, a different grading scheme (see Chapter 2, Teamwork) requires the academic coaches to assess skills and products that traditionally are not evaluated.

In order to ease the academic coaches' adaptation to the new coaching environment, it is recommended that students in the first semester of the transition be offered projects that have already been developed in previous semesters, and that the scale of such projects be adjusted to team sizes of ten to twelve students. This way the academic coaches can focus on the new coaching environment without being distracted by technical issues. In the following semesters, when the coaching team feels more comfortable with the new framework, new topics for projects can be introduced.

In Chapter 12, Change, we describe a two-day workshop which aims at presenting the main ideas of agile software development in a condensed format. It is explained how this format can be adjusted for the training of a team of academic coaches. In Chapter 13, Leadership, we present an academic coaching framework.

Task

Envision a studio organization in which a team of about ten students develops a software project by an agile process. Explain each of your decisions. What values did you attempt to convey with this organization? How would you implement them?

1.8.5 Overview of the Studio Meetings

The studio enables a kind of real-world work environment with some adaptations to academia. During the fourteen studio meetings, a software project is developed in an agile process, which is composed of three iterations. Each project is worked on by a team of ten to twelve students, guided by an academic coach. If possible, it

is preferable to provide each team its own studio, equipped with furniture, computers, and whiteboards.

The different agile practices are introduced gradually so that, on the one hand, they do not overwhelm the students; but on the other hand, they do indeed address all aspects of agile software development. In addition, practices that require high cognitive awareness (such as refactoring and test-driven development) are revisited several times during the semester. In this way, students can come to an understanding of these practices in stages, based on their current and updated experience gained in the process of project development. This is consistent with the constructivism perspective, which ascribes a significant role in learning processes to the student's experience (see Teaching and Learning Principle 2, above, and Chapter 7, Learning).

The first iteration deals with the topic of the project and with software engineering issues. It enables the students to study related issues. In this iteration, the team's spirit and atmosphere are created. This aspect gets high importance, especially in academia, since the team members are usually not familiar with each other prior to this course. With respect to the development itself, the purpose of the first iteration is to produce an outline for the project so as to enable additions and improvements during the second and third iterations.

By the time of the second iteration, the learners are more relaxed and more familiar with the development method and with their teammates, and most of them invest the effort to increase and improve their project outcomes. Most of the meetings are dedicated to development activities. Features and improvements are added to the project in this, as in this and the third iteration.

In order to achieve the main goals of the studio, attendance at studio meetings is mandatory, and it is important to arrive on time.

During later stages of the course the students gradually become more familiar with the details of the studio and the activities that take place in it.

1.8.6 Launching the Project Development in the Studio

The main objective of the first studio meeting is to let the students become familiar with the studio as the development environment. For this purpose, the course structure and the studio work environment are introduced, teams are formed, the project subjects to be developed by the students are presented and distributed among the teams, the development schedule is explained, and the students meet the customer(s).

1.8.6.1 Project Description

Each of the project subjects is described to the students in one paragraph and the lecturer then elaborates on them, giving additional details and answering students' questions. The customer(s) is introduced and the required collaboration with the customer is emphasized.

Examples of project descriptions are presented in Table 1.4. As can be seen, the project descriptions are very general and do not include many details. This is done on purpose, to let the readers experience project descriptions the way customers sometimes provide them in the early stages of the development process, when they are not sure what their specific requirements are. Such general descriptions, however, serve as a good opportunity to ask the customer clarification questions in order to improve the teammates' understanding of the project to be developed.

Tasks

1. Read the project descriptions presented in Table 1.4. For each, present three questions that you would ask a customer who presents the description.
2. Select a software system. Try to present it in three or four sentences in a similar manner to the ones presented in Table 1.4. Propose at least three questions that, in your opinion, a teammate may ask when presented with your description.

Table 1.4 Examples of one-paragraph project descriptions

Subject	Description
A web-based survey system	In this project we will develop a web-based survey system. On the one hand, the system enables the surveyor to define the survey questions, to determine how the questions are processed, and to view the results. On the other hand, the system should enable the people who respond to the survey to fill it in
A virtual machine	In this project we will develop a virtual machine (VM). A VM is an abstract machine that enables the running of a code. The VM enables parallel running of many threads. Its main components are the memory manager, the executable engine, and the thread package manager. In this project, we will also develop a VM GUI for end users
A traffic control system	In this project we will develop a traffic control system. It enables one to observe the main crossroads in a defined area and to change the traffic light length accordingly. It also includes several control patterns so that inspectors are able to make fast decisions during rush hours

1.8.6.2 Team Forming

After the project descriptions are presented, teams are formed based on the students' preferences, diversity considerations, and course resources. Figure 1.3 presents an example of a form that can be used for this purpose. One form is filled out by two or three students, thus enabling each student to select one or two other students with whom they wish to be on a team. At the same time, however, in order to let the students become familiar with other course participants, and in this way to expose themselves to new perspectives and people and to educate them to be open to diverse teams, the number of students that each student can choose to be on the same team with is limited to one or two.

<University, Department>

<Course name>

Forming Software Teams

Two or three students who wish to belong to one team are requested to fill in the following form:

Student's name	Student's ID	Department or Faculty	Electronic mail

Project priorities

First priority: _____

Second priority: _____

Third priority: _____

Figure 1.3 *Form for creating teams.*

After the students fill in these forms, student teams should be created taking into consideration their preferences with respect to teammates and project subject, team diversity in order to make sure that teams will not be too homogeneous, and other issues which are relevant for specific course conditions and the teaching staff's pedagogical preferences.

Task

If you had to select one or two students to be in your development team, what would be your considerations?

1.9 Summary and Reflective Questions

1. What are your expectations from the course? Refer both to the lectures and to the studio components of the course.
2. What topics would you expect to learn in the course? Refer both to the lectures and to the studio components of the course.
3. What skills would you expect to acquire in the course? Refer both to the lectures and to the studio components of the course.
4. Explore several agile methods presented at the Agile Alliance website: http://www.agilealliance.org/library. What are their main characteristics? In what ways are they similar? In what ways are they different? How does each of them apply and reflect the Agile Manifesto?
5. From your review of the Agile Alliance website, what can you learn about the agile software development community?
6. Brooks, in his famous article "No Silver Bullet," published in April 1987 in *Computer* magazine, argued that "there is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity." How does this perspective relate to agile software development?
7. What other approaches to software development exit? In what ways are they similar to agile software development? In what ways do they differ from agile software development?

1.10 Summary

This chapter aims at inspiring the spirit of agile software development. It outlines the Agile Manifesto and explains how it is applied by agile software development methods. Several agile practices are also sketched in this chapter; most of them will be elaborated in the next chapters of the book, when we explore the different facets of agile software development, such as quality, time, learning, and management. In addition, this chapter describes the studio and outlines the main activities carried out in the first studio meeting. These activities aim at forming teams and establishing the physical conditions for the actual software project development to be launched in the next meeting.

References

Davis RB, Maher CA, Noddings N (1990) Constructivist views on the teaching and learning of mathematics. J Res Math Educ Monograph Number 4, The National Council of Teachers of Mathematics, Inc.

Dubinsky Y, Hazzan O (2005) A Framework for teaching software development methods. Comput Sci Educ 15(4):275–296

Hazzan O (2002) The reflective practitioner perspective in software engineering education. J Syst Software 63(3):161–171

Hazzan O, Dubinsky Y (2003) Teaching a software development methodology: the case of extreme programming. In: Proceedings of the 16th international conference on software engineering education and training, Madrid, Spain, pp 176–184

Hazzan O, Dubinsky Y (2006) Teaching framework for software development methods. Poster presented at the ICSE Educator's Track, Proceedings of ICSE (International Conference of Software Engineering), Shanghai, China, pp 703–706

Hazzan O, Dubinsky Y (2007) Teaching agile software development quality assurance. In: Stamelos I, Sfetsos P (eds) The agile software development quality assurance book. Idea Group Inc., pp 171–184

Kuhn S (1998) The software design studio: an exploration. IEEE Software 15(2):65–71

Smith JP, diSessa AA, Roschelle J (1993) Misconceptions reconceived: a constructivist analysis of knowledge in transition. J Learn Sci 3:115–163

Tomayko JE (1996) Carnegie-Mellon's software development studio: a five-year retrospective. SEI conference on software engineering education http://www.contrib.andrew.cmu.edu/usr/emile/studio/coach.htm.

Williams L, Kessler R, Cunningham W, Jeffries R (2000) Strengthening the case for pair-programming. IEEE Software (July/August) 17:19–25

2
Teamwork

Abstract

This chapter presents one of the basic elements of software projects—teamwork. It addresses how to build teams in a way that promotes team members' accountability and responsibility, and that fosters communication between teammates. One of the basic ways to start team building is by assigning roles to the team members. For this purpose a role scheme is presented in this chapter, according to which each team member is in charge of a specific managerial aspect of the development process, such as design and continuous integration, in addition to his or her development tasks. Teamwork is not always a simple process, and sometimes it raises dilemmas and conflicts between team members. This aspect of teamwork is not neglected in agile teams, and when a conflict emerges, it is addressed openly by all the team members. In the section that deals with teamwork in learning environments, it is illustrated how the role scheme and the discussion about dilemmas in teamwork provide an evaluation framework for software projects developed by student teams in academia.

2.1 Overview

This chapter focuses on teams—one of the main and most influential factors of software projects' success. Consequently, it is highly appreciated and supported by agile software development methods.

One practice which is highlighted in this chapter is applying a role scheme, according to which each team member has an additional role in the team in

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_2, © Springer-Verlag London Limited 2008

addition to being a software developer. This role scheme fosters the interconnections and dependencies between the members of agile teams and enhances creativity, responsibility, accountability, diversity, and measure collection. Further, this role scheme shows that each team member can contribute to software development on the team level, beyond his or her individual contribution, and that the mutual contributions of the individuals in the team create a whole which is greater than the sum of its parts.

To highlight the importance attributed to teamwork in agile software development environments, this chapter also presents a set of activities that deal with roles, which may form the needed atmosphere for agile teamwork.

In addition, we discuss potential dilemmas in teamwork and how agile development may help cope with such dilemmas. Specifically, we examine how to satisfy the individual needs of each team member and, at the same time, achieve the needed contributions to the team's work.

Based on this examination, an evaluation scheme for student software projects, developed in teams, is presented in the section of this chapter that deals with teamwork in learning environments. This evaluation scheme takes into consideration the various conflicts that may arise.

2.2 Objectives

- Readers will become familiar with the characteristics of software teams in agile software development environments.
- Readers will learn how to allocate roles to members of agile teams and how to exploit the benefits of role assignment at the individual, team, and organization levels.
- Readers will discuss dilemmas in teamwork and understand how agile teams can overcome them.
- Readers will get a sense of how agile team spirit can be achieved, empowered, and maintained.
- Readers will gain basic skills to exploit the strength of agile teams.

2.3 Study Questions

1. Find definitions for the concepts of team and teamwork. Discuss the definitions' relevance for software teams.
2. What is the purpose of teams? Why are teams needed?

3. As a software team member, how would you like to feel in your team? What values would you like to pursue? What practices would you want to see in your team development environment to provide an atmosphere that fits you?
4. What roles do you have in your life? How do you manage to perform them all? Describe two personal scenarios in which a conflict emerged from the need to play different roles and explain how the conflicts were solved.
5. Why should roles be assigned to software team members? What roles would you assign to your team members?
6. Look through the literature for different approaches to role assignments in software teams in general and in agile software teams in particular. Compare these approaches: What ideas do they share? How do they differ from each other? What are the specific characteristics and responsibilities of each role holder in the team?
7. Discuss at least three problems which software teams face. Suggest ways to solve these problems.
8. Suggest dilemmas which software team members might face. Suggest ways to solve these problems. Suggest agile principles and practices that can support such cases.

2.4 A Role Scheme in Agile Teams

According to Humphrey (2000), a team consists of at least two people who are working towards a common goal/objective/mission, in which each person has been assigned a specific role to perform and in which a completion of the mission requires some form of dependency among team members (p. 19). In the case of a software project, a team is a group of individuals who have gathered to produce a software product.

In software projects teams are needed for the accomplishment of the complex task of software development. It is however, not a trivial task to mange software teamwork. This is partially because software development is about an intangible product, one that cannot be seen, smelled, or touched, and therefore, the development status and the exact responsibilities are not always clear.

The unique situation of software development can be approached in different ways. We will start with role assignment, which is one way by which agile software development methods attempt to overcome the typical challenges of software projects. Role assignment means that each team member has an additional role besides that of developer.

The assignment of roles serves as a means for splitting the responsibility for project management and progress among all the team members. The rationale for this stems from the fact that one person (or a small number of developers) cannot manage the entire richness and complexity involved in software development projects. When the responsibility is split among all teammates, each aspect of the process is treated by one teammate, and each teammate feels a personal responsibility for that specific aspect. Both the software project as a whole and each of the individual team members benefit from this kind of organization.

Table 2.1 presents a role scheme for an agile software team, which expands and integrates the role schemes suggested by the different agile methods. It is based on the idea that the responsibility for the software's progress and success should be transferred to and distributed among *all* teammates (Dubinsky and Hazzan 2004, 2006).

There are four groups of roles.

The first is the *leading group,* which consists of the coach, tracker, and methodologist. It is important to note that the tracker and methodologist in this group, as well as the other role holders, do not reduce the significance of the project leader/coach. To the contrary—as it turns out, the role scheme improves the project leader's position and provides him or her a better way to assess and lead the development process.

The second is the *customer group,* which consists of the user evaluator, customer, and acceptance tester. If the project has a real customer, the roles in the customer group should be conceived of as a bridge between the real customer and the team. If the project does not have a real customer, these role holders play a real customer.

The third group is the *code group,* which is composed of four roles: designer, unit tester, continuous integrator, and code reviewer. This group of roles focuses on those aspects of software development that are directly related to the coding activity.

The fourth group is the *maintenance group,* which comprises three roles: presenter, documenter, and installer, and focuses mainly on the product's external presentation and documentation.

As can be seen, the different roles address different aspects of the development process (leadership, customer, code, and maintenance), and together they encompass all the aspects of a standard software development process.

Holding a role, and thus being in charge of a specific aspect of the development process, *does not* mean that the role holder performs all the activities related to his or her particular role; instead, it implies that each role holder must ensure that the specific aspect for which he or she is responsible will be carried out properly by all team members. Since this idea applies to all team members, the project management is split among all team members and, at the same time, is covered by all of them.

Table 2.1 Roles in an agile software team (Reprinted from Journal of System Architecture, 52, Dubinsky Y, Hazzan O. Using a role scheme to derive software project quality, 693–699, Copyright (2006), with permission from Elsevier. Also, with kind permission of Springer Science and Business Media.)

Group of roles	Role	Description
Leading group	Coach	Coordinates and solves group problems, leads and guides development sessions
	Tracker	Measures the group progress by measures as defined by the team, the customer, and the organization; manages the workspace boards; manages the team diary/collective memory. See also Chapter 5, Measures
	Methodologist	Makes sure that the team works according to the defined development process, answers questions related to the methodology, looks for solutions to problems related to the methodology
Customer group	User evaluator	Performs an ongoing user evaluation of the product (collects and processes feedback received from real end users), holds a user centric approach, serves as the user interface designer. See also Chapter 3, Customers and Users
	Customer	If the project doesn't have a real customer: tells customer stories, makes decisions pertaining to each iteration, provides feedback, defines acceptance tests. See also Chapter 3, Customers and Users
	Acceptance tester	Defines (with the customer) and develops acceptance tests, inspires a test-driven development process. See also Chapter 6, Quality
Code group	Designer	Maintains current design, works to simplify design, searches for refactoring tasks and ensures their proper execution. See also Chapter 8, Abstraction
	Unit tester	Establishes an automated test suite, guides and supports others in the development of unit tests, guides a test-driven development process. See also Chapter 6, Quality
	Continuous integrator	Establishes the integration environment; publishes and encourages rules pertaining to the addition of new code, including testing issues
	Code reviewer	Maintains source control, establishes and refines coding standards, guides and manages the team's pair programming
Maintenance group	Presenter	Plans and organizes iteration/release presentations, demos, and roles; measures presentations
	Documenter	Plans and organizes the project documentation: process documentation, user's guide, and installation instructions
	Installer	Plans and ensures the development of an automated installation kit, maintains the collaborative workspace infrastructure

For example, let us assume that one of the teammates is a developer who also has the role of unit tester. As the unit tester, this team member is in charge of all the unit testing activities of the entire project, and of guiding other teammates in the development of their unit tests. But this does not paint the entire picture: let's look at this team member from the perspective of being a developer. As a developer, this team member should write quality unit tests for his or her own development tasks. As the person who is specialized in unit testing, it makes sense that she or he will write quality tests, which can in turn serve as examples for the other teammates of how unit tests should be written. In addition, as a developer, this team member is guided with respect to other aspects of the development process by team members who hold the other roles. This scenario shows how the two hats of each team member—a role holder and a developer—are interconnected and contribute to the development process, improve the software quality, and reinforce the team members' communication.

Task

In a way similar to the analysis presented above with respect to the unit tester role, analyze at least three additional roles from Table 2.1.

The cumulative impact of all the roles increases the team members' commitment to the project. In order to carry out a role successfully, each team member must gain a global view of the developed software and to be involved in all parts of the application, in addition to carrying out his or her personal development tasks. If a team member has a limited view and is aware only of his or her tasks, he or she will not be able to perform the personal role properly. The need to accomplish the personal role satisfactorily actually increases one's involvement and accountability, as well as the commitment to the development process, and leads one to become familiar with all the software parts. Consequently, communication and knowledge sharing, which are vital for software development, are once again increased among team members.

In general, in having a personal role, the team members are expected to perform their development tasks as well as the tasks related to their personal role. Thus, no teammate is merely a developer. The two activities have a mutual positive influence, and consequently the collaboration and communication between the team members is enhanced and the agile team's spirit is maintained. Further, the dual functionality of each team member increases the transparency of both the software process and the product. The process becomes more transparent because it is clear who is in charge of each of its aspects; the product becomes more transparent because each team member is familiar with all the product components and aspects, at least with respect to his or her role.

This perspective is different from the approach in which each role holder is responsible for the entire implementation of the aspect of which he or she is in charge, while the other team members do not perform it at all (e.g., the documenter is the

only team member who is involved in documentation activities). While this approach may lead other team members to reduce their responsibility with respect to that activity, the role scheme described above just inspires the opposite: a strong interconnection exists between the teammates. One team member cannot properly perform all the tasks involved in a software project, even with respect to only one aspect of the development process; cooperation between teammates is essential in order to accomplish the development process properly and on time. Further, since the role scheme clarifies the exact responsibilities of each team member, team members are committed to each other by ensuring that all of them are able to perform their roles in the best way. All these messages delivered by the role scheme increase the team members' involvement and commitment to the project and to the team.

To sum up: Roles are important for the establishment and maintenance of agile teams. A clear role scheme inspires an agile spirit and contributes both to the individuals, to the team, and to the project's success. Further, the role scheme spreads the leadership and management of the software project among all the team members. It lets the developers know that not all the responsibility is carried by one person.

2.4.1 Remarks on the Implementation of the Role Scheme

- The set of roles that a software development method includes in its role scheme reflects the values that a software development method attempts to inspire. Therefore, the role scheme that a software development method defines is one of the key elements of the method. Indeed, different agile methods suggest different role schemes that support their values and conception of software development processes.
- In addition to the role definitions presented in Table 2.1, several roles also support communication between the four groups. For example, the installer is also in charge of communication with the code group.
- At the first stages of the development process, or when the team is established, the role holders should learn their roles and establish a procedure that will enable them to perform their role properly. In the next stages of the agile project, role holders should maintain the spirit and the actual performance of the aspect that their role focuses on.
- When teams consist of fewer than twelve developers, several roles can be unified and assigned to one team member. There are different ways to unify roles, and each has its own advantages. In each case, however, the entire list of roles should be assigned and performed by all team members.

- The team can choose whether each of its members will specialize in one role for a long period of time or, alternatively, whether the roles will rotate among the team members. The exact way by which it is done in practice should be set by each team according to the team members' preferences. For example, it can be decided that roles are reassigned at the beginning of each release.
- Such a team organization eases project management, since it is clear who is in charge of what aspect of the development project, what aspect should be treated by whom, and who should be approached when a specific problem, which belongs to a specific aspect of the development process, arises. Even in cases when there are role overlaps, they will not interrupt the process. Sometimes they can even foster project development. One example is when the unit tester and the acceptance tester work together to introduce test-driven development.

Tasks

1. How does the role scheme reflect the HOT perspectives of agile methods?
2. Predict what attitudes and feelings such a role scheme might raise in agile software teams.
3. What information does each role holder need in order to perform his or her role successfully?
4. In what ways does the role scheme relate to the Agile Manifesto?
5. Describe how each of the Agile Manifesto principles is supported by the role scheme.
6. What benefits does role rotation have?
7. Suggest a mechanism for role rotation in agile teams. What are its benefits? What are its pitfalls?

2.4.2 Human Perspective on the Role Scheme

Social Aspect

- A personal role increases teammates' involvement, communication, accountability, responsibility, and commitment to the software development process and to their team.
- Team members wish to have a specific role in addition to their development tasks in order to increase their influence and involvement in the project management.

Cognitive Aspect

- Since each team member approaches the product from one specific perspective, each can focus on this one specific aspect without being distracted by the multifaceted nature of software product development. In other words, on the global level, the role definition encourages each team member to treat the software product from one perspective. Consequently, each gradually improves his or her understanding about that aspect.
- The role scheme supports the thinking of the development process on multiple levels of abstraction (see Chapter 8, Abstraction). Since abstraction is a key component of software development, every mechanism that supports team members' thinking in terms of different levels of abstraction should be enhanced. On the one hand, each team member sees his or her development task on a relatively low level of abstraction; and on the other hand, the personal role of each team member enables each of them to gain a global overview of the developed system on a higher level of abstraction. Agile methods support thinking at different levels of abstraction in additional ways, such as short releases (see Chapter 3, Customers and Users) and refactoring (see Chapter 8, Abstraction).
- The role scheme enhances knowledge distribution, since each team member specializes in one domain and shares his or her knowledge with the other team members. In addition, since the role scheme leads to knowledge distribution, no harm happens when one team member leaves the team. Indeed, he or she has gained expertise in his or her role; at the same time, however, parts of this knowledge have already been spread. Thus, if a team member leaves the team, the other team members have a reasonable amount of knowledge to continue with respect to that role.
- The role scheme supports the individual's professional development. Team members perform their roles and improve their role performance while learning the practice that their role represents. In turn, they become experts in the specific aspect of software development on which their personal role focuses. In addition, when a team member feels that he or she has exhausted one role's contribution to his or her professional development and wishes to hold another role in the team, as has already been mentioned, role rotation can take place.

Tasks

1. To each of the ideas presented in the human perspective on the role scheme, add its organizational and technological view.
2. Analyze the role scheme from the organizational and the technological perspectives.

2.4.3 Using the Role Scheme to Scale Agile Projects

The role scheme also supports the scaling up of agile projects. Suppose we have five agile teams as part of one software project, and each of them applies the role scheme. In this setting, weekly role meetings are set for each role, in which all the role holders from all the teams participate. For example, a weekly meeting of all testers of the project takes place; a biweekly meeting of all the integrators takes place, etc. It is recommended that these role meetings be scheduled at the same time, in order not to collide with the development sessions of the project teams. In these meetings project-wide issues are discussed, so that the project management proceeds in one direction.

The use of the role scheme for scaling up purposes also enhances knowledge distribution. On the individual level, each team member has the opportunity to communicate with other developers, beyond his or her team, to present the knowledge his or her team has gained so far with respect to a given role, and to serve as a bridge between the team and the organization with respect to that aspect of development of which she or he is in charge. On the team level, each team may benefit also from the wisdom and experience gained by other teams. For example, the team representatives may bring into the role meetings a problem which their team faces, and ask the other role representatives whether their experience can contribute to a solution. Such a dialogue creates a knowledge infrastructure for the development process from which all teams can benefit. On the organization level, and based on the individual and team levels, knowledge is distributed, managed, and maintained.

The role scheme also supports measures related to the project's progress (see Chapter 5, Measures). The measures and policies that should be applied by all the teams enable the project's management to know on an ongoing basis the project's status, progress, and quality. Based on this information, management monitors and controls the project's progress.

2.5 Dilemmas in Teamwork

One of the problems that can arise with respect to teamwork is the question of how to allocate incentives, rewards, and bonuses among team members.

This question is relevant with respect to many professionals and kinds of institutions. However, reward allocation in software engineering is important mainly, but not only, because teamwork is essential in software development. As a result, conflicts between the required cooperation on the one hand, and one's desire to excel as an individual on the other, may intensify. The discussion is

especially relevant with respect to *agile* teams since teamwork is one of the basic working assumptions of agile software development, and team members are asked to cooperate, share information, and exchange ongoing feedback with the other players in the development environment.

Task

This task is based on Hazzan (2003). It aims at elevating the developers' awareness to these potential conflicts and to encourage discussing them openly. This approach is in agreement with the first principle of the Agile Manifesto: individuals and interactions over processes and tools.

Perform the task presented in Figure 2.1 with your team.

Step 1 of the task focuses on the individual's preferences; step 2 examines how team members face possible conflicts between their own preferences and the preferences of the other team members. Thus, in the case of new teams, this activity also fosters the team members' acquaintance with each other.

The discussion that takes place at step 3 focuses on the team preferences at the individual and at the team level. This discussion can be promoted by the following reflective questions.

Step 1: Individual work

You are a member of a software development team. Your team is told that if the project it is working on is successfully completed on time, the team will receive a bonus. Five options for bonus allocation are outlined below. Please explain how each option might influence team cooperation, and select the option you prefer.

	Personal Bonus (% of the total bonus)	Team Bonus (% of the total bonus)	How this option may influence teammates' cooperation
A	100	0	
B	80	20	
C	50	50	
D	20	80	
E	0	100	

Step 2: Teamwork (to be facilitated with the development team)

Each team decides on one option that all team members, as a group, prefer.

Step 3: All teams discussion

Discuss with all the teams the processes that took place in the above two steps.

Figure 2.1 *Bonus allocation activity [© 2003 ACM, Inc.].*

Reflective Questions

1. What were your considerations when choosing your personal option for bonus allocation?
2. Did you face conflicts while working on this task individually (Step 1)? What was their source? How did you overcome these conflicts?
3. Did you face conflicts while working on this task with your team (Step 2)? What was their source? How did you overcome these conflicts?
4. What questions, emotions, and dilemmas with respect to software teams were raised during individual and team work?
5. Predict what considerations would cause developers to prefer a different option for bonus allocation than yours.
6. What characterized the discussion in your team about the agreed upon option for bonus allocation? How did the team agree about the preferred option?

2.6 Teamwork in Learning Environments

The studio meeting this week focuses on activities related to the introduction of the role scheme, the role assignment, and the grading policy that is used for the evaluation of the students' work. The details appear in the continuation of this section.

2.6.1 Teaching and Learning Principles

The following teaching and learning principle deals with the role scheme. (In our list of teaching and learning principles presented in Chapter 14, Delivery and Cyclicality, this is principle number 7.)

Teaching and Learning Principle 7: Assign Roles to Team Members.

According to this principle, each team member has both an individual role, chosen by the member from a given list (for example, coach, unit tester, acceptance tester, code reviewer, etc.), and development tasks for which he or she is responsible.

Such a role scheme does not imply that each role holder carries out all activities related to the domain for which he or she is responsible; rather, each role holder makes sure that the activities related to hir or her domain are accomplished satisfactorily by all team members. Accordingly, the assignment of roles helps divide the responsibility for project progress and management among all team members.

The rationale for this principle is that one person (or a small number of team members) cannot be responsible for the entire richness and complexity involved in software development. When the responsibility is divided among all team members, each aspect of the entire process is addressed by one team member, and at the same time each team member feels personal responsibility for that specific aspect. Both the project itself and the team members benefit from this arrangement. Furthermore, the need to perform one's role successfully actually forces all the team members to be involved in, and to become familiar with, all parts of the developed application. Consequently, knowledge sharing, communication, and involvement are enhanced among team members.

2.6.2 Role Activities

We present the actual application of the role scheme through three kinds of activities. The first kind deals with the role assignments. Second, activities that maintain the role performances on a daily basis are described. Third, an activity that aims at improving the role performances is presented. The activities can be performed in both academic and industrial settings.

2.6.2.1 Role Assignment Activities

The first two activities introduce the role scheme to the team members. If the activities are carried out in an industrial setting, they should be facilitated when the agile team is established, in order to let the team members feel the interconnection among themselves, and their mutual responsibility as an agile software development team. The other activities should be facilitated as development proceeds.

Figure 2.2 describes the first activity related to role assignment. It focuses on the creation of one agreed upon role list.

Since in this studio meeting the academic coach sits together with ten to twelve students who do not know each other but will soon start working together on many tasks related to software development, this activity initiates the students' relationships as teammates.

Time: In academia: second meeting; in industry: when an agile method starts to be implemented.
Task description: You are going to develop a software product as a team. Write down the roles that in your opinion should be performed as part of your project.
Individual work (10 minutes): Students/developers write down their lists.
Team discussion (20 minutes): Students/developers discuss their suggestions, trying to generate one agreed upon list.
For students: Students are asked to prepare a prioritized list of roles they would prefer to perform during the semester.
Summary: The academic coach/agile facilitator presents the role scheme (Table 2.1).

Figure 2.2 *Activity 1: role list generation (Reprinted from Journal of System Architecture, 52, Dubinsky Y, Hazzan O. Using a role scheme to derive software project quality, 693–699, Copyright (2006), with permission from Elsevier.).*

In industry, this activity signals the beginning of a change in the team structure with respect to personal responsibilities. It can be facilitated when the team is first introduced to agile software development, as part of a workshop that the team attends (see Chapter 12, Change) or at the beginning of the implementation phase of agile software development.

In both cases, Activity 1 improves the team members' acquaintance of and familiarity with their teammates.

Activity 2 (Figure 2.3) describes the role distribution.

In academia, the full set of roles is determined for the entire semester. This full implementation is needed for the evaluation process, presented later in this chapter, which is based on the fact that all students have the same load.

Also, according to the role scheme presented in this chapter, the students are responsible for the software's progress and success. Therefore, it is important to note that the academic coach should *not* be the team coach, and that the role of coach should be given to one of the students. The academic coach is in charge of

Time: In academia: second meeting; in industry: following the previous activity.
Discussion (10 minutes): In your opinion, how should we assign roles to teammates?
Task description (20 minutes): Distribute the roles among the teammates. At the end of the task suggest a list of role-teammate pairs.
Discussion (15 minutes): Discuss what portion of your time you should dedicate to the performance of your personal role and what portion should be devoted to the accomplishment of your development tasks?
Reflection (after the second meeting in academia; during a team discussion in industry):
- Express your opinion about the process of role assignment in your team.
- How do you conceive of your role? What input would you expect to get from your teammates with respect to your role?

Figure 2.3 *Activity 2: role distribution (Reprinted from Journal of System Architecture, 52, Dubinsky Y, Hazzan O. Using a role scheme to derive software project quality, 693–699, Copyright (2006), with permission from Elsevier.).*

project evaluation and control from the academic perspective, but does not lead the actual development process. This perspective gives the academic coach a better way to assess the development process and the teammates' work by means of the grading policy, presented later in this section, that supports the role scheme and is based on both an individual component and a team component.

In an industrial setting, the role scheme should be applied in a gradual fashion. It is recommended that the team decide with what roles they would prefer to start the agile implementation phase. The team chooses several roles to start with, and team members volunteer to carry out these roles. It is recommended that the team start at least with the roles of coach, tracker, and unit tester, and gradually add roles according to the team preferences, needs, and adjustment to the agile process.

The actual role assignment itself has a direct influence on team communication. For example, in an industrial setting, it opens new horizons to team members: they are exposed to new facets of their teammates of which they were not aware, even though they may have worked together for many years.

This activity also influences and enhances learning. For example, team members may suggest that roles should be assigned not according to what is appealing, but rather according to what area one is *not* skilled in. In this way, in order to perform their role properly team members will have to communicate about the various aspects of their role with other teammates who are more knowledgeable. Thus the team members will learn new topics.

2.6.2.2 Role Maintenance Activities

The activities in this part are performed on a regular (daily, weekly, iteration) basis. In academia this enables the academic coach to be aware of the project's progress and to improve the students' work assessment; in industry it enables the entire team to be aware of the project status.

Stand-up meeting: Stand-up meetings take place every day in industry, and on a weekly basis in academia. It can be decided that some portion of the brief personal report (one or two sentences) be dedicated to the personal role. Each team member reports about his or her role performance and about his or her expectations from teammates with respect to personal roles.

Presentations to customers: The following task fits for an academic setting; when appropriate, it can be adjusted for an industrial setting. Specifically, each presentation to the customer consists of two parts. In the first part, the development tasks of the iteration are presented; in the second part, each student briefly presents how he or she improves product quality by the accomplishment of his or her role.

The preparation for these presentations takes place one week before the presentation of each iteration to the customer. The students can discuss what the presentation should contain, how much time will be dedicated for each part, and how the personal

roles will be presented. Since students usually do not have previous experience in presenting software products, they will benefit a lot from this activity.

Feedback after presentations: In academia, there are three presentations to the customer during the semester: at the seventh, eleventh, and fourteenth (the last) meetings (see Chapter 1, Introduction to Agile Software Development, for a semester schedule). After each presentation two reflective activities are carried out: the first is a feedback session that takes place with the all team members; the second is a personal reflection that encourages the students to evaluate their work in the last iteration. As part of this reflection, students are requested to evaluate their role performance as well as to grade it.

2.6.2.3 Role Improvement Activity

The following activity can be requested by the academic coach periodically when she or he observes that it is needed. Such a need occurs mainly in cases when the academic coach feels that some roles are not being performed properly. The students are asked to summarize their role activities, to publish their summaries in the electronic forum, and to provide feedback to the summaries presented by the other team members.

2.6.3 Student Evaluation

One outcome of the team discussions that take place in the bonus allocation activity (Figure 2.1) is an agreement that each individual's contribution should be considered on the basis of his or her personal accomplishments (no matter how the team performs), as well as the team performance.

Accordingly, in an academic setting, if the course is accompanied by a software development project, we propose to make the project evaluation of each student be based on a personal evaluation (independently of how the team performs), and on an evaluation of the team performance. In order to promote teamwork as well as personal contributions to the project, the two components should be balanced in some way. For this purpose, the students' evaluation is composed of two parts: one the personal component and one the team component. The evaluation of the individual role performance is used for the personal evaluation.

The proposed evaluation scheme, presented in Table 2.2, applies these conclusions. It is composed of an individual component (35%) and a team component (65%); the team component is identical for all members of the team. The main criterion of the individual component of the grading scheme is the personal performance of the student (50%) as well as his or her personal role (25%). The main criterion of the group component is the presentation of the customer stories as well as the time estimations given by the students at each of the three development iterations.

Table 2.2 Grading policy

Individual component (35%)	Group component (65%)
50%—	**60%**—
✓ Weekly reflection	Answer the customer stories and meeting the schedule according to the group time estimations:
✓ Pair programming experience	✓ **(10%)** for iteration 1
✓ Test-Driven-Development exercise	✓ **(25%)** for iteration 2
✓ Weekly presence	✓ **(25%)** for iteration 3
25%—	**25%**—
Performance of a personal role:	Project documentation
✓ Actual implementation	**15%**—
✓ Further development and enhancement	Group evaluation by the academic coach
25%—	
Personal evaluation of the academic coach	

This student evaluation scheme shows that both teamwork and individual contribution count. Accordingly, it is assumed that on the one hand, students will be encouraged to contribute to their team's work, and that on the other hand, those wishing to excel will have the opportunity to improve their grade through the personal component. Consequently, the personal responsibility of each student is increased.

The grading policy presented in Table 2.2 also fosters gradual improvement. This is accomplished by the main parts of each grade component. In the group component, 60% of the grade is achieved by meeting the customer requirements according to the students' own estimations. The first iteration (out of the three iterations) receives the lowest portion (even though it lasts half a semester), thus demonstrating that this iteration is dedicated to learning about the environment, the teammates, the method, and the project. In the individual component, about 75% of the grade is achieved by presence and reflection, for specific exercises, and for performing the personal role. These parts are also characterized by gradual learning. Students are more open to give feedback as the semester proceeds, and role performance is improved after the learning iteration, which is the first iteration of the semester.

Tasks

1. In your opinion, should different kinds of teams be evaluated in different ways?
2. Should teams be formed of students with similar preferences, with different preferences, or according to a specific combination of preferences, with respect to bonus allocation? What considerations should guide each of these options?
3. In what ways is bonus allocation similar to course grading? In what ways is it different?

2.7 Concluding Reflective Questions

1. What were your considerations when choosing a personal role? Why?
2. What were your teammates' considerations when choosing a personal role? Why, in your opinion, were these their considerations?
3. How will you accomplish your role successfully?
4. What are the three main goals of agile software teams?
5. In your opinion, what are the three most important characteristics of agile software teams? How do these characteristics enable them to achieve their goals successfully (Question 4)?
6. In Chapter 9, Trust, we will meet two ideas related to agile teamwork: diversity and ethics. If you are familiar with these concepts, suggest connections between them and the way agile teams are built and function.

2.8 Summary

This chapter introduces the first steps of agile teams by establishing their structure and some of their development habits and processes. This is done by assigning personal roles to team members, which on a personal level improves their understanding of the developed product, and on the team level improves the process and product quality. The role scheme that guides the role assignment achieves these goals by defining personal roles and cross-project roles; that is, each role holder is responsible for the management of a specific aspect throughout the *entire* project.

This chapter also introduces dilemmas in agile teamwork and how they can be addressed. In the spirit of the Agile Manifesto, dilemmas and conflicts associated with teamwork should be discussed openly by all team members, and a solution that meets the needs of the individuals as well as the entire team should be established. This idea has been illustrated by an evaluation scheme for student projects.

References

Dubinsky Y, Hazzan O (2004) Roles in agile software development teams. 5th international conference on extreme programming and agile processes in software engineering. Garmisch-Partenkirchen, Germany, pp 157–165

Dubinsky Y, Hazzan O (2006) Using a role scheme to derive software project quality. J Syst Architect 52 (11): 693–699

Hazzan O (2003) Computer science students' conception of the relationship between reward (grade) and cooperation. 8th annual conference on innovation and technology in computer science education (ITiCSE 2003), Thessaloniki, Greece, pp 178–182

Humphrey W (2000) Introduction to the team software process. Addison-Wesley, Reading, MA

3
Customers and Users

Abstract

Customers have a key role is agile software development processes. The Agile Manifesto statement "customer collaboration over contract negotiation" explicitly suggests alternative work relationships with the customer, in which the customer is part of the team, rather than an entity with whom team members and management argue about what should be developed and when, and what should not be developed at all. This chapter describes the customer role in agile development environments and how to enhance communication with the customer. In many cases, the customer also represents the user group and is expected to evaluate the software product from the user perspective. This, however, turns out to be an ineffective mechanism in many cases. Accordingly, this chapter also describes the users' role and presents a mechanism to increase their involvement and feedback during the development process. Specifically, data obtained from user evaluations is used to refine the user interface design, to increase product usability, and, consequently, to increase product quality.

3.1 Overview

This chapter presents the perspective of the customers and users for whom the software product is developed.

The agile approach to software development emphasizes "individuals and interactions" in the process of software development (See the Agile Manifesto in

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_3, © Springer-Verlag London Limited 2008

Chapter 1, Introduction to Agile Software Development). When software developers are asked who these individuals are, most of them would probably mention different roles like system analysts, developers, and testers. The agile approach increases the awareness of additional essential roles in the development environment—for example, the customer, who is one of the most important project stakeholders. The users, at the same time, are somehow wrongly neglected. A common misconception is that the customer represents all users. In this chapter these two roles are distinguished and the customer and user roles are described by addressing their main responsibilities in agile software development.


HOT

The customer. The customer's position and role in the software development environment is one of the main changes that the agile approach introduced into the process in general and into team members' conception of the customer role in particular. This customer position in agile software development is central. It is based on ongoing communication between the customer and the team members, both with respect to the requirements, as well as with respect to the way testing is performed and how the suitability of the developed product to the customer's needs is achieved. This communication is established in several ways. Among other things, we focus in this chapter on the practice of planning and on the concept of a common language; both foster customer-teammate communication and bridge the gap (if one exists) between the customer's and the teammates' worldviews. As it turns out, the customer role is not only supported by several practices, but also fosters characteristics of agile software development, such as information sharing. One of the main ideas delivered in the customer section of this chapter is that the agile approach supports the customer role and enables the required collaboration needed for the production of high quality products.

The user. While the customer is one of the few people who either actually pays for the software development or has other kinds of interests in the development process, in the context of most software projects, the users form the main clients. This is where the world of agile software development meets the world of human computer interaction (HCI). HCI adds the user perspective to agile software development by its offering of user evaluation methods and by its implications for the software design and development processes (Dix et al. 2004, Hwong et al. 2004, McInerney and Maurer 2005, Norman 2006, Rogers et al. 2002).

In other software development approaches, when user-centric techniques are used, the system is refined according to user evaluations, mainly during the design phase. In agile environments, the design is shaped during the entire development process and no specific upfront phase is set for this purpose. Accordingly, questions such as the following are raised when HCI approaches are addressed in agile software development environments: How can user-centric techniques be implemented in agile software development? How can the two worlds—HCI and agile software development—be merged?

One of the main ideas delivered in the user section of this chapter is the combination of and the mutual connections and contributions between agile development and HCI. Specifically, on the one hand, the user evaluation is fostered by the agile process; on the other hand, the product development benefits from keeping the software design updated according to the ongoing user evaluation.

3.2 Objectives

- Readers will get acquainted with the customer and user roles and responsibilities.
- Readers will become familiar with the notions of customer collaboration and user-centric development.
- Readers will learn about specific practices that enable customer collaboration.
- Readers will learn about user evaluation activities.
- Readers will get acquainted with how agile software development supports the customers' and the users' involvement in decision making processes as well as in design and evaluation activities.

3.3 Study Questions

1. Search for and present quantitative and qualitative data about the customers' role and position in software development.
2. What is customer collaboration and how can it be achieved?
3. In your opinion, what should be the customer's duties?
4. In your opinion, what should be the customer's privileges?
5. What mechanisms does the agile approach provide to assist and support customer collaboration and user involvement?
6. Would you like to serve as a customer of a software product? Why?
7. Search for project descriptions, not necessarily software engineering projects, with high user involvement. What are their main benefits? What are their main drawbacks?
8. What is a user-centric approach? Why should users be involved in software development?

3.4 The Customer

3.4.1 Customer Role

Let us speculate what the consequences would be of the following interactions in software development environments.

Software teams tend to declare that customers do not know what they need. System analysts, as well as developers, feel the urge to explain to customers their (the customers') requirements. Such interactions occur during the development process and cause an ongoing negotiation of the software requirements. Usually, this negotiation is not pleasant and accompanied with harsh feelings on both sides. Customers feel that they are not authorized to decide what the requirements are. In some cases, customers feel that they should either fight for what they wish to get, or just give up and compromise. Developers feel that customers are not certain with respect to their requirements, and therefore believe that they can decide about the requirements.

Not surprisingly, in many such cases (in about 75% of big software projects), the customers do not use the software at all, or alternatively, need to compromise with what they get (Mullet 1999). In addition, software projects fail to accomplish on-time delivery. For example, according to the Standish Group Chaos Report (Standish 1994), "over one-third [of projects] ... experienced time overruns of 200–300%. The average overrun is 222% of the original time estimate. For large companies, the average is 230%; for medium companies, the average is 202%; and for small companies, the average is 239%."

The conception of the customer role in agile software development is different. It is significantly extended and receives a new interpretation. This is not limited to merely listening to the customer; rather, it also implies that the customer's decisions will be followed. This concept can be implemented because the customer is involved in the development process continuously, as is presented in what follows.

A project schedule comprising short releases of three to four months each is set. Each release includes short iterations of one to a few weeks. See Figure 3.1 for an example of two releases of three months each with two-week iterations.

As part of release planning, the following activities take place:

- The customer describes the project vision, the project main stories, and the guidelines according to which development priorities will be set.
- The architects present their vision about the product architecture, including the existing architecture and anticipated changes.

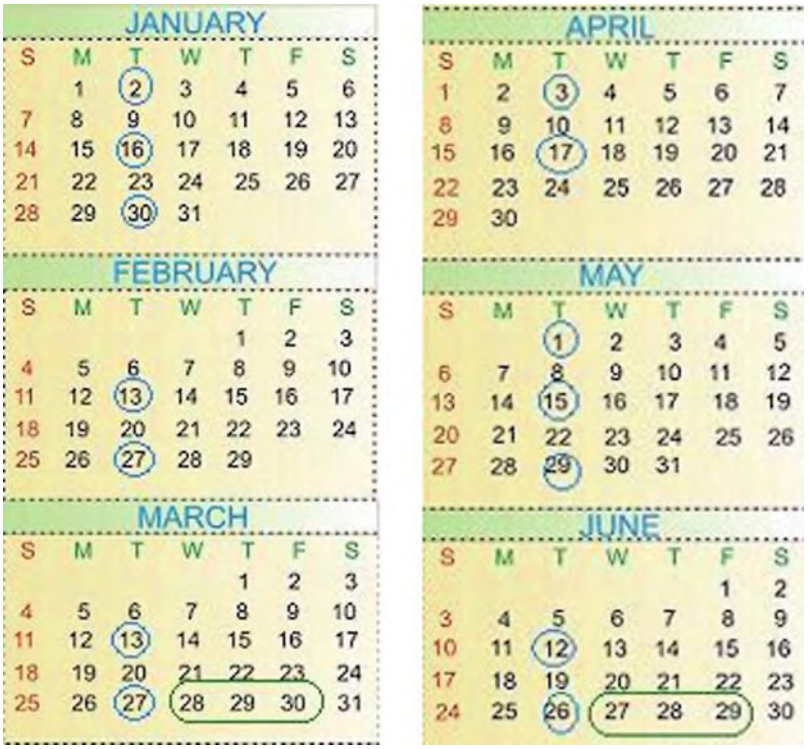


Figure 3.1 *A two-release calendar of 2007 with marked business days.*

- The project manager presents his or her view of the development process and the working environment as well as his or her personal expectations.
- Other stake holders present their expectations from the development process.

This part of the release planning takes place after the presentation of the previous release has been completed and a retrospective session between the two releases has been facilitated (see Chapter 11, Reflection, and Chapter 14, Delivery and Cyclicality).

The development tasks of each iteration are determined in a planning session which is part of a Business Day (Dubinsky et al. 2005a). A Business Day takes place between each two consecutive iterations of the release: at the end of each iteration and the beginning of the next one (for example, between the forth and the fifth iterations). The rest of the iteration days are development days.

For example, in Figure 3.1, the project road map is based on two three-month releases and includes a Business Day every two weeks (circled). In between two

Business Days, development days take place. At the end of each release a retrospective (see Chapter 11, Reflection) and release planning take place.

A Business Day Between Two Iterations

A Business Day takes place between iterations and between releases.

On the Business Day, in addition to the team and the customer, other project stake holders are invited to participate, including managers and external parties such as customer associates and users.

It is common not to perform Business Days on the first working day or the last working day of the week in order to eliminate the pressure to work over weekends.

In the first part of the Business Day the previous iteration is summarized. In the second part, after a reflective session takes place, the next iteration planning starts. The Business Day between iterations is time-boxed to one working day and the exact schedule of the different activities may vary between projects; in most cases, the planning starts after the lunch break.

The main Business Day activities are:

- presentation of the accomplishments of the ending iteration;
- measures' review;
- customer feedback;
- reflective session;
- planning of the next iteration.

During all these activities the customer has a significant role, as is explained in the following descriptions of each element of the Business Day.

The **presentation of the system** demonstrates the main new features developed in the ending iteration, executed on the actual integrated system. The features that are presented belong to specific customer stories of the ending iteration. A customer story being defined in Beck and Fowler (2000) as follows: "The story is the unit of functionality in [a] ... project. We demonstrate progress by delivering tested, integrated code that implements a story. A story should be understandable to customers and developers, testable, valuable to the customer, and small enough so that programmers can build half a dozen in an iteration" (p. 45). The customer business interest is also emphasized: "The most important stories to do first are the ones that contain the highest business value. Beware of sequencing stories based on technical dependencies. Most of the time, the dependencies are less important than the value" (p. 63).

In many cases, the planning session does not replace a separate meeting with the customer in which the entire iteration's product is presented. This is done

towards the end of each iteration, and normally takes two to four hours of a task-by-task demonstration and intense freehand use of the system by the customer. The Business Day presentation has two different goals. The first is to make sure that the system is indeed fully integrated, and includes all the customer stories of the ending iteration, and that the team is able to deploy it when needed. The second is to make sure that everyone knows all of the system's features, from the customer's and the user's (in contrast to the programmer's) point of view. The second reason becomes prominent as the deployment approaches.

During the system presentation each team member presents his or her work as a developer (see Chapter 2, Teamwork). This activity raises the developers' accountability in two ways. First, they should develop high quality code that answers a specific customer story, is supported by tests, and is fully integrated into the system. Second, they should present this output at every iteration in front of all people who are interested in the development, including managers and external parties as well as their own team. Since each team member shares the information with all the other people involved and answers their questions, the overall understanding of the project components and features is increased.

The **measures review** is a presentation and analysis of the ending iteration's metrics. The following four metrics are interesting for many agile teams (Dubinsky et al. 2005b). The product metric (the number of written and passed tests), the pulse metric (a measure of continuous integration [Beck 2000, see also Chapter 5, Measures]), the burn-down metric (an estimation of the convergence to the release/iteration goals), and fault metrics (the number of new and open defects). Further elaboration about these metrics can be found in Chapter 5, Measures.

The goal of this element of the iteration summary is twofold. First is to present the data to the entire team, replacing individual perceptions (for example, about product quality, time lost to overhead, etc.) by facts. Second is to openly discuss the reasons behind the metrics, since metrics cannot be analyzed regardless of context. For example, a decrease in the number of new defects does not necessarily stem from improved product quality: It may be the case that less testing was performed in this iteration (and thus fewer bugs were found), or that people did not report all defects into the common defects table (for example, if they fixed them at once and consider the report redundant).

The customer role in this part is first, "to be there," which highlights the fact that both the process and progress are transparent and that the customer is updated continuously about the project status. The decisions taken by the customer are also influenced by the information that is provided in the measures presentation. For example, if the measures indicate that a specific component is more complex than it seemed when planned, then the customer can decide to change the development scope and/or the priorities. Second, the customer can

enhance collaboration by adding measures of his or her own interest. This way the teammates improve their understanding with respect to the customer emphases and priorities.

Task

What, in your opinion, are the most important topics to highlight in the system presentation and the measures review?

The **customer feedback** is a short, informal verbal summary of the iteration, given by the customer. This direct feedback usually focuses on the product rather than on the process. It is important to include the customer's message in the iteration summary to signal the customer's importance in the development process. It also helps in focusing people on the product as an end goal, rather than their own specific tasks or the development process.

The **reflective session** is intended to discuss a specific issue in the development process, and to change the process if needed. This practice is described in detail Chapter 11, Reflection. This part of the Business Day is announced and considered as a timeout, a time to stop considering the regular, mainly technical, issues and to think about other kind of topics. Usually, people enjoy this timeout, cooperate in bringing new issues to the discussion, and volunteer to take responsibility to follow up on things that they find interesting and relevant for them.

Task

In your opinion, what are the topics to be highlighted in the customer feedback and the reflective session elements of the first part of the Business Day? Compare your ideas with your colleagues' opinions. Are your opinion and interest similar to you colleagues? Are they different? What, in your opinion, is the source of these differences or similarities?

Planning the next iteration starts immediately after the previous iteration is summarized and its reflective sessions ended. As in the first part of the Business Day, the customer and all team members participate; other people who have interest in the project are invited.

First, the customer tells the stories that were prepared in advance to be developed in the next iteration. To the customer list, stories from other sources are added, such as: incomplete stories from previous iterations, refactoring tasks (see Chapter 8, Abstraction), and user interface stories that emerged from the user evaluation (see the second part of this chapter). The customer role in this part is to prioritize the stories so that all the people involved, including all the

team members, hear and realize what, from the customer perspective, the important stories and the less important stories for this iteration are.

Second, based on earlier work that in some cases is done by system analysts and in other cases by architects and team leaders, the top prioritized development tasks are described. Then the development time of each of these tasks is estimated by the developers who take ownership of them. The actual planning is set according to the available time of the team members in the coming iteration.

The fact that the developers estimate the development time is important. This practice replaces development environments in which developers are frustrated, since schedules are set without considering their time estimation. In many such cases, project and team leaders decide upon requirements and schedules together with the customer. Then the team leaders serve as the customer, answer the developers' questions according to their understanding of the project, and decide upon priorities. They also distribute the work among the team members. Since such scenarios increase the developers' frustration, the agile approach attempts to avoid them by letting the developers take the responsibility for time estimation and task allocation.

Finally, the development loads are balanced among the developers.

The iteration planning is performed differently by different agile methods; still, a few principles are maintained:

- Time is an important resource and should be managed wisely.
- The smaller a development task is, the more accurate its development time estimation is. Thus, product delivery on time and of high quality is better ensured.
- An ordered professional work environment is required by professional practitioners; chaos frustrates professionals especially because the resulting products are of low quality and their professionalism is doubted.
- Fairness and a cooperative work environment are valued by professional developers; an open and transparent work distribution, in which all parties are involved, increases practitioners' security, trust, and cooperation.

The planning subject is elaborated on Chapter 4, Time.

Tasks

1. What are the main customer responsibilities?
2. What are the main characteristics of the Business Day? For each characteristic, specify which HOT—Human, Organizational and Technological—perspective you use.

3. In light of the Business Day description, what, in your opinion, are the customer's feelings during a Business Day? Would you like to serve as the customer in the Business Day format? Why?
4. How is the Agile Manifesto expressed in the above description of the customer role?
5. Describe how the HOT perspectives on software engineering is expressed in the Business Day.

3.4.2 Customer Collaboration

As one of the Agile Manifesto principles states, customer collaboration is one of the most significant issues of agile software development, and it plays an important role in agile teamwork. This section focuses on the common language required in order to maintain ongoing communication with the customers. Specifically, we describe the use of metaphors (Beck 2000).

Metaphors are used in order to understand and experience one specific thing using the terms of another (Lakoff and Johnson 1980, Lawler 1999). Communication which uses metaphors to improve our understanding of a specific thing refers not only to instances in which both worlds of concepts correspond to one another, but also to cases in which they do not. If both worlds of concepts are identical, the metaphor is not a metaphor of a thing, but rather the thing itself.

For example, Beck (2000) suggests "driving" as a metaphor for software development. The process of software development is controlled by the execution of many small adjustments in a similar way to car driving. Feedback is required when the driver is slightly "off," and many opportunities are needed for corrections at a reasonable cost. This metaphor, however, is limited. For example, no teamwork is involved in driving. Still, this metaphor is worthwhile.

Several attempts to use metaphors in agile projects have been made, but all have reported difficulties in adopting this practice (Gittins and Hope 2001, Wilson 2001, Johnson and Caristi 2001). Indeed, metaphor is a practice that requires a high level of cognitive awareness when implemented (Hazzan and Dubinsky 2003). Here are two guidelines that can be used when metaphors aim at increasing and improving communication in software projects.

First, *be aware of metaphors*. We all use metaphors naturally, and from those metaphors we can learn about our own as well as others' (teammates, customers) understanding of a given system. This understanding can support our communication. In most cases, when developers see the horizons that metaphors can open, they respond positively and want to extend their use. This leads to the need to talk about metaphors and to encourage their use.

Second, *encourage developers to provide multiple metaphors:* the more metaphors we use, the more we improve both our understanding of the project and our communication with respect to the various aspects of the project, as well as with respect to difficulties the project stakeholders might encounter during the course of project development. No harm arises when using several metaphors for different parts of the project, using each one according to need. Developers are willing to use different metaphors for the same topic, and it does not interfere with the project's progress. On the team level, multiple metaphors increase understanding, mainly because each metaphor supports the thinking of different team members, and in many cases, when there are several metaphors, they complement each other.

Tasks

1. What metaphors can you suggest for the projects on which you currently work?
2. Suggest a metaphor for the notion of software development. Discuss similarities and differences between the metaphor and the notion of software development.
3. Lakoff and Johnson (1980) explain the metaphor "Argument ***is*** War." Suggest three illustrations for this metaphor. Use this metaphor to describe typical situations that may occur with customers in software projects.
4. Suggest a metaphor to describe the kind of customer collaboration which the agile approach inspires.

3.5 The User

The human computer interaction (HCI) field emerged in the early 1960s. It deals with interface design and evaluation and with the interactions between users and systems, where systems can be hardware, software, or both. The main goal of the HCI field is to improve these interfaces and interactions according to the users' needs. This is done by rigorous techniques that involve users and HCI design experts in the design of the user interface and the evaluation process.

Norman (2006) suggests abandoning the traditional HCI approach of "study first, design second" and to try the "design, then study" approach. This suggestion is influenced by the agile approach to software development and therefore is included in this book.

Task

1. In what way does the "design, then study" approach fit the agile approach?
2. In your opinion, in what way can the user contribute to the customer's role in agile software development?

Since usability is "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use" (ISO 9241–11, 1998), it is essential to integrate the users in the development process. This importance is highlighted when the most useful indicators in measuring the usability level of a product, as defined by the ISO 9241 standards, are examined:

- Effectiveness in use, which encompasses accuracy and completeness, through which users achieve certain results.
- Efficiency in use, which has to do with the resources utilized in relation to accuracy and completeness.
- Satisfaction in use, which includes freedom from inconveniences and a positive attitude toward the use of a product.

Tasks

1. Chose a software tool you use on a regular basis. Suggest specific examples for each of the above indicators.
2. Suggest specific examples for each of the above indicators with respect to the software project you currently develop.

The integration of the user in the development environment is accomplished by the user centered design (UCD) approach, which is a set of design techniques that emphasizes user needs during the design of the user interface. This is achieved by managing user evaluation with validated user evaluation techniques (Vredenburg et al. 2002).

Evaluation of user interfaces aims at assessing the extent of system functionality while the user interacts with and gains experience with the system, and identifies specific problems related to the system (Dix et al. 2004). There are two main types of evaluation: expert-based evaluation and user-based evaluation.

In **expert-based evaluation**, a designer or an HCI expert assesses a design of user interfaces based on known cognitive principles or empirical results. The **user-based evaluation** is based on user participation, i.e., evaluation that involves the people who are going to use the system. User-based evaluation techniques include experimental methods, observational methods, questionnaires, interviews, and

physiological monitoring methods. User-based evaluations can be conducted in a laboratory and/or in the field.

Surprisingly, it is known that the best evaluation results come from small tests with no more than five users, conducted in several iterations (Nielsen and Landauer 1993). Therefore, the evaluation process is not an expensive process, as it is sometimes wrongly conceived. For example, Nielsen and Landauer (1993) describe an iterative design in which the evaluation of five users revealed 85% of the usability problems. Accordingly, the design of the user interface was changed and reevaluated to ensure that problems were fixed and that new problems had not emerged. Indeed, reevaluation iterations probe deeper usability problems.

Tasks

1. For a specific software project with which you are acquainted, define the users and explain when and how they should be involved in the project development.
2. Choose a software application that you use on a regular basis. Ask the development team of this application how to get you involved in the development process as a user.
3. In your opinion, what should be the relationship between the users and the customer? Between the users and the development team?

3.5.1 Combining UCD with Agile Development

The UCD approach goes hand in hand with the agile software development approach and they mutually benefit each other (Blomkvist 2005); hence users should be constantly involved in the software development process. User evaluation contributes to the set of measures used for the steering and directing of agile projects (see Chapter 5, Measures) and also enhances the emerging design of the user interfaces which are part of the software project.

Case Study 3.1 illustrates the combination of UCD with agile software development.

3.5.1.1 Case Study 3.1. Merging Development Iterations with User Evaluation Iterations

This case study illustrates the combination of UCD with agile software development in a specific software project called Catalogue Browsing Project (CBP) in

which a speech-based mobile interface to a naïve digital library was developed (Dubinsky et al. 2007). The main goal of the CBP development project is to provide a seamless interaction between the physical and digital realms in accessing library artefacts, based on the concept of browsing through a catalogue. An artefact of a digital library can be accessed over the web, and it contains a metadata record with descriptive details like title, author, year of publication, etc., as well as the data itself.

The first release of CBP was developed by two people during four months (from the middle of May till the beginning of September 2006) and divided into four iterations of about three to five weeks. Customer collaboration and user evaluation were emphasized during the development process. Details about the implementation and some CBP screenshots can be found in Dubinsky et al. (2007).

The CBP evaluation process was composed of evaluation iterations, each one examining the artefacts of the previous development iteration and resulting in design changes for the next iteration. Sometimes, there were changes in the current iteration, especially when these changes were small, related to the stories of the current iteration, and ranked by the users as high severity. In this sense, this can be seen as a bug fix that was found and fixed in the same iteration.

The stages of this process—development and evaluation—are described in what follows.

1. The 1st development iteration provides its artefacts.
2. During the 2nd development iteration, the 1st evaluation iteration takes place to evaluate and reflect on the artefacts developed in the 1st development iteration and to decide what changes should be introduced, if any, into one of the next development iterations, according to the decisions made in the planning session.
3. During the 3rd development iteration, the 2nd evaluation iteration takes place to evaluate and reflect on the artefacts produced in the 2nd development iteration, and so on.

Task

How is this process related to agile software development?

In the first two iterations the user groups were identified. They included librarians and readers. Questionnaires and semistructured interviews were prepared in order to improve the understanding of users' needs.

In the third iteration an evaluation was performed with two users. The purpose of this evaluation was to learn about the users' interaction with the system and about any major problems encountered.

After the fourth iteration had ended, that is, after the first release was over, an experiment for speech evaluation was planned and conducted. Besides the system evaluation, the main goal was to provide guidelines for the evaluation of speech-based user interfaces to ensure better design of these interfaces.

An experiment with six participants was conducted. The six users were computer science students in different stages of their studies, three male and three female. The experiment tasks included login to the system, search activities, and a book localization activity. The task could have been performed with a speech interface (S) or without a speech interface (non-S). Each of the participants performed the task in both modes (S and non-S), while three participants started with S and proceeded to non-S and three started with non-S and continued to S. In addition, before the experiment had started, each participant filled out an attitude questionnaire with respect to speech interfaces and received a ten-minute CBP usage training. Each of the participants performed the experiment separately. After the experiment had ended, each participant filled in a reflective questionnaire on his or her activities.

In what follows the experiment stages, together with qualitative and quantitative data, are presented.

Pre-experiment stage: This stage was carried out before the users experienced the system.

In general, it was observed that the attitudes towards speech interfaces were mixed and did not reflect a consistent approach. Though the speech interfaces were found to be fun, they were also annoying; and though participants liked them, they did not always prefer them.

Experiment stage: After the participants had filled in the questionnaire, they received a one-page users' guide for the CBP and were asked to read it and to ask questions.

Then the users received the relevant task page according to their experiment order of S and non-S. As mentioned previously, each of the participants performed the experiment separately. An *automatic* time measure, which was developed as part of the system, provided the login/logout time stamps as well as the time stamps for each search start/end.

Table 3.1 presents the average time in minutes invested in the two search activities by both experiment groups, together with their duration per mode.

Table 3.1 Average search time (in minutes) (With kind permission of Springer Science and Business Media.)

Group	Average search duration	Average Non-S search duration	Average S search duration
Non-S → S	54.66	28	81.33
S → Non-S	26.58	14	39.16

As can be observed, the S→Non-S group performed the entire task almost twice as fast as the Non-S→S group. When looking into the data of speech and non-speech per group, it can be observed that the participants in both groups performed the speech task more slowly than the non-speech task. This implies that although the speech task required a longer time, the participants learned the system in a better way when they used it first with the speech option.

Task

What can be learned from this observation?

Post-experiment stage: After completing the CBP task, the participants were asked to reflect on their own activities. Some participants found it hard to use CBP in its current development stage, though it was fun and they expected such interfaces in the future.

In addition, users' answers revealed that some improvements are needed. These improvements deal with speech implementation for all interface features and with online usage information. This conclusion was based on the observation that when users are introduced to a speech-based interface they expect it to be fully speech-based i.e., not using the keyboard at all. Further, they expect to receive vocal online help to assist them in the process of using the application.

Tasks

1. Suggest specific tasks that should be implemented in the next development iteration that are directly derived from the above user evaluation.
2. What lessons could have been learned from each of the experiment stages?
3. Had you been assigned to interview the users, what questions would you have asked them?
4. Had you been assigned to observe how the participants used the system, what elements would you have focused on?
5. How can customers use the information obtained from user evaluations? What benefits would customers gain from learning and taking into the consideration user evaluations?
6. Describe the main agile ideas expressed in Case Study 3.1.

3.6 Customers and Users in Learning Environments

3.6.1 Teaching and Learning Principles

The following two teaching and learning principles deal with communication and establish a common language by using metaphors. In our list of teaching and learning principles, presented in Chapter 14, Delivery and Cyclicality, these principles are numbers 5 and 10.

3.6.1.1 Teaching and Learning Principle 5: Elicit Communication

Communication is a central theme in software development. Indeed, the success or failure of software projects is sometimes attributed to *people* communication issues. Accordingly, in all learning situations, aim at fostering learner-learner, as well as learner-teacher communication.

This idea is implemented in many ways. One option is to present students with conflicts and problems in software teamwork. The opportunity to discuss sensitive topics face-to-face has direct implications on students' teamwork in general, and their cooperation and decision-making processes in the team in particular. Another process which elicits communication is the retrospective, to which we dedicate a lot of attention.

3.6.1.2 Teaching and Learning Principle 10: Use Metaphors or "Other World" Concepts

This principle addresses the cognitive aspects of software development. Generally speaking, metaphors are used in order to understand and experience one specific thing using the terms of another thing (Lakoff and Johnson 1980). Metaphors are used naturally in our daily life, as well as in educational environments.

Metaphors can be useful even without specifically mentioning the metaphor concept. For example, a teacher may say, "Can you suggest another conceptual world that might help us understand this unclear issue." Learners then suggest a varied collection of conceptual worlds, each highlighting a different aspect of the issue and together supporting the comprehension of the topic discussed.

3.6.2 Customer Stories

In this studio meeting the teammates meet the customer and listen to the customer's vision and the stories that are expected to be developed in the coming release. If the project either does not have a customer at all or has more than one customer, a discussion about who is the customer takes place and a proxy customer is nominated, usually one of the students.

Specifically, during the meeting, teammates are exposed to the entire list of customer stories, and hear how the customer prioritizes them. During the prioritization process, the academic coach explains that in the next iteration things can be changed by the customer according to the presentation of the artefacts developed and the resulting new understandings. This kind of behavior is encouraged, since the goal is to provide the customer with what he or she needs. Further, the customer is asked to elaborate with details about who the users of the system are; and the user evaluator (a student role) is asked to prepare a work plan, together with other teammates, for the purpose of the users' involvement.

When the list of stories for the first iteration is ready, the academic supervisor guides an open discussion, its aim is to improve the understanding of each story. The customer is asked to explain unclear matters, and the students are requested to make sure that they understand what is needed.

At the end of this meeting, the team is familiar with the stories to be developed in the first iteration. They are asked to spend the next week preparing a high level design according to these stories and a list of development tasks to be distributed among teammates. It is explained that this list of stories is not necessarily the exact work that will be developed in the first iteration, and that the exact scope will be determined in the next meeting when the development tasks will be estimated and compared to the available time. If the estimated time turns out to be longer than the available time, the customer will decide what stories to omit; and if the estimated time turns out to be shorter that the available time, the customer will decide what stories to add.

3.6.3 Case Studies of Metaphor Use

To illustrate how metaphor can be used to foster customer-teammates communication, we present several case studies of agile projects developed by students. For each case, the project and the use of metaphor with respect to its development are described.

3.6.3.1 Case Study 3.2. Identification of Short Sequence Repetitions in a DNA Sequence

This aim of this project was to identify short sequence repetitions (SSR) in a DNA sequence. A graphical interface was provided, and after receiving a DNA sequence

and several parameters about the required repetition, the system was required to return the following: the basic components of the repeated element, the number of repetitions of that element in the sequence, the length of the repeated segment, and the starting position of the segment within the DNA sequence.

The students were requested to provide metaphors for the project's subject. These metaphors are presented in Table 3.2. In this case, the metaphors were also needed by the lecturer and the assistant in order to enhance their own understanding of the biological subject. Discussing these metaphors with groups of students helped the lecturer and the assistant promote their understanding of the various concepts of the project.

Task

Do the metaphors presented in Table 3.2 improve your understanding of the project?

3.6.3.2 Case Study 3.3. Personal Information Organizer

This project dealt with a personal information organizer that acts as an interface between the user and different applications that enable access to the user's personal data. The system contains a navigator that provides effective data management, and enables the formation of dependency relationships between different kinds of data, which can be reached by the user from within different kinds of applications.

For example, the user can create a dependency between specific data in a document file and specific data in a spreadsheet file. A uniform format is used for all files in order to enable dependency relationships. In addition, the system provides a special security feature that resembles a private virtual safety deposit

Table 3.2 Metaphors for the SSR project

	Biology terminology	Metaphoric expressions
Teammate # 1	The DNA sequence	A sequence of 0s and 1s as electrical pulses of a computer
	The SSR	It contains sequences of 0s and 1s
Teammate # 2	The DNA sequence	A topographic road map with slopes, traffic signs, traffic lights, etc.
	The SSR	Contains repetitions of sequences
Assistant	The DNA sequence	A sequence of bits received when listening to the network by a communication card
	The SSR	The repetition of new packet bits

box which is implemented for all users using client/server architecture. A graphic monitor is provided for viewing general virtual vault transactions. In addition, the monitor enables each user to view personal security information, such as when and by whom each file was accessed.

Students were given specific instructions concerning protocols, databases, programming languages, and development techniques that were to be used in the implementation of this system.

During the first planning sessions, after listening to the customer stories, most students were uncertain about how to think about the file navigator. The customer emphasized that he was not comfortable with the current tree hierarchy of files, which he found to be poorly organized. According to the client, files were saved in incorrect locations, rendering them impossible to retrieve. As a result, such files did not undergo automatic synchronization with other related files.

Task

What metaphors would you suggest for this system? How would they help you communicate the nature of the system?

One of the students suggested the navigator be treated as if it were an *association graph*. An association graph is made up of nodes that represent specific subjects, and connecting arcs, which denote equivalent relationships between the nodes. The idea was to regard each such node as representing a file, and each arc between two files as denoting a first-degree relation with respect to the file contents. The student explained that the use of such an association graph to manage the files would enable the user to manage each file according to his or her own perception of the relationships between a given file and other files. Thus, a graph could be created in which adjacent nodes represent files that are more strongly related to one another than nodes that are more distant. The student concluded his explanation by stating that this is the way in which our minds work. Some students claimed that this scheme would be impossible to implement. However, during a discussion with the academic coach, in which the students used the above metaphor to explain how our minds relate to data, they began to realize how the project could be planned using the metaphor.

Eventually, the "file organization ***is*** mental association" accompanied the group's discussions throughout the semester. Following are several examples of this metaphor, as expressed by the students:

- "After the user creates a file he will decide how to save it according to *its relation to* other files."
- "The file will be *associatively* close to another file if their contents are related."

- "The distance between one file and another can be *infinite*."
- "At any given moment we can ask which files have *a level-2 relation* to a specific file."
- "The file tree should be suited as much as possible to the user, and each user is an individual, and the subjects are integrated differently in each *individual's mind*."

Tasks

1. In what sense do the above statements support communication among team members and between the team and the customer?
2. In what ways, if any, does the proposed metaphor improve your understanding of the system?
3. Can you add statements that reflect how the "file organization ***is*** mental association" metaphor can be used?

3.6.3.3 Case Study 3.4. Simulator of the Unix™ File System Module

The subject of this project was an educational computer program that simulates the work of the Unix™ file system module. The system relates to the following subjects: buffer cache, file system structure of inodes table and data block, and the superblock for management of available inodes and data blocks. Each file is represented by an inode in the inodes table, and each inode contains file details such as owner and permissions, as well as a pointer to the data blocks that contain the file's contents. The system has a graphical interface that enables the user to learn about the file system structure, perform actual training, read online help, and view running simulations of Unix™ commands.

Task

What metaphors would you suggest for this system?

The students were requested to provide metaphors for the project's subject as a whole or for its parts. The proposed metaphors are presented in Table 3.3.

Table 3.3 Students' metaphors for the Unix™ file system educational computer program

Project terminology	Metaphoric expressions
Creating a GUI	Painting a picture
Simulation screen in our project	A transparent phone that enables us to see how the system works inside
The project as a whole	– A piece of land on which we begin to build rooms, and then add details inside the rooms – A vehicle that has many parts and together they enable it to move
Files permissions in the Unix™ file system differ between file owner, the group, and others	At the bank, the clerk has permission to look at the accounts. The bank manager and the client have other permissions

Tasks

1. In what sense is each of the descriptions presented in Table 3.3 a metaphor for the developed software?
2. In what ways, if any, do these metaphors improve your understanding of the system?
3. Can you add metaphors for this system?
4. In what sense may the above metaphors support communication among team members and between team members and the customer?

The students were asked about the main problems in the software. One of the students talked about the difficulty of understanding how to implement the inodes table in general, and about the connection between each inode and the data blocks in particular. The students discussed this problem and mentioned the metaphors "an inode ***is*** a book record" and "file data blocks ***are*** a book." The first of these two metaphors, as explained by the students, refers to the situation in which a specific book is found in a library catalogue. The book record contains details such as the author and publisher, as well as a pointer to the location of the actual book in the library. The idea behind the second metaphor was found to be irrelevant as students continued to discuss it, since file data blocks in Unix™ are not consecutive, unlike the pages of a book.

Following this discussion, the students were asked about the contribution of the discussion to their understanding of the problem and to its solution. Following are the responses of several students:

- "The discussion using the metaphor helped open an additional channel by which to understand. . . ."

- "The metaphor about the catalogue and library shelves illustrated well how a file search is performed when we know its inode number."
- "The truth is that it helped me personally, since it made me realize that I didn't really understand it before."
- "Actually, in the end, we didn't use the metaphor. . . ."
- "The metaphor changed the concepts from being distant to being tangible. . . ."
- "I think that had we started this subject with the metaphor we would have understood it faster."
- "The discussion using the metaphor didn't contribute to my understanding, but didn't bother me either, because I understood the subject even before the discussion. . . ."
- "The library idea is perfect, but in my opinion the right metaphor in order to understand . . . is a family tree. . . ."

Tasks

1. What benefits does a discussion have that leads to a decision that a specific metaphor does not fit for the said system?
2. How do metaphors enhance understanding when problems are discussed?

3.7 Summary and Reflective Questions

1. Indicate three main ideas that you learned about the customer's position and role in agile software development.
2. Indicate three main ideas that you learned about the users' role in agile software development.
3. During the development process of your last project, what were the customer's main stories? Why, in your opinion, were these requirements prioritized in this way?
4. During the development process of your last project, were the customer's stories clear? What were your main questions with respect to the customer's stories?
5. What is your opinion with respect to the way teammates hear the project requirements? How is this similar or different from your personal experience?

6. Suggest a work plan to involve users in the project you are currently working on.
7. What are the main concepts learned in this chapter? How are they connected to each other?
8. How is each of the concepts reviewed in this chapter connected to agile software development? Answer this question from the HOT perspectives.

3.8 Summary

In this chapter the customers' and the users' roles in agile software development are described. The activities by which the customer navigates the project by telling the development stories, prioritizes the stories, and gives ongoing feedback to the teammates with respect to the developed artefacts, are laid out. This kind of collaboration provides the setting needed for dealing with change requests, thus establishing a process that leads to the development of a high quality product from the customer's perspective.

User involvement in software projects in order to develop the needed user interface is also discussed in this chapter. For this purpose, a user-centered approach is adopted for the interface design and evaluation.

References

Beck K (2000) Extreme programming explained: embrace change. Addison-Wesley, Reading, MA

Beck K, Fowler M (2000) Planning extreme programming. Addison-Wesley, Reading, MA

Blomkvist S (2005) Towards a model for bridging agile development and user-centered design. In: Seffah A, Gulliksen J, Desmarais M (eds) Human-centered software engineering—integrating usability in the development process. Springer, Dordrecht, The Netherlands

Dix A, Finlay J, Abowd GD, Beale R (2004) Human-computer-interaction, 3rd ed. Scotprint, Haddington

Dubinsky Y, Hazzan O, Keren A (2005a) introducing extreme programming into a software project at the israeli air force. Proceedings of the 6th international conference on extreme programming and agile processes in software engineering. Sheffield University, UK

Dubinsky Y, Talby D, Hazzan O, Keren A (2005b) Agile metrics at the Israeli air force. Agile conference, Denver, CO

Dubinsky Y, Catarci T, Kimani S (2007) A user-based method for speech interface development. In: Stephanidis C (ed) Universal access in HCI, part I, HCII, LNCS 4554, HCI international, Beijing, China. pp 355–364

Gittins R, Hope S (2001) A study of human solutions in extreme programming. 13th annual workshop of the psychology of programming interest group. pp 41–51

Hazzan O, Dubinsky Y (2003) Bridging cognitive and social chasms in software development using extreme programming. Proceedings of the 4th international conference on extreme programming and agile processes in software engineering, Genoa, Italy. pp 47–53

Hwong B, Laurance D, Rudorfer A, Song X (2004) User-centered design and agile software development processes. Identifying 2004, workshop bridging gaps between HCI and software engineering and design, and boundary objects to bridge them. CHI workshop, Vienna, Austria

ISO 9241–11 (1998) Ergonomic requirements for office work with visual display terminals: guidance on usability

Johnson DH, Caristi J (2001) Extreme programming and the software design course. XP/Agile Universe website

Lakoff G, Johnson M (1980) Metaphors we live by. University of Chicago Press

Lawler JM (1999) Metaphors we compute by. In: Hickey DJ Figures of thought: for college writers. Mayfield Publishing

McInerney P, Maurer F (2005) UCD in agile projects: dream team or odd couple? Interactions 12(6):19–23

Mullet D (1999) The software crisis. Benchmarks Online—a monthly publication of Academic Computing Services 2(7): http://www.unt.edu/benchmarks/archives/1999/july99/crisis.htm

Nielsen J, Landauer TK (1993) A mathematical model of the finding of usability problems. Proceedings of ACM INTERCHI'93 conference, Amsterdam, The Netherlands, pp 206–213

Norman D (2006) Why doing user observations first is wrong. ACM Interact July-August 50ff

Rogers Y, Preece J, Sharp H (2002) Interaction design: beyond human-computer interaction. Wiley, New York

Standish Group Chaos Report: http://www.standishgroup.com/sample˙research/chaos_1994_2.php

Vredenburg K, Isensee S, Righi C (2002) User-centered design: an integrated approach. Software Quality Institute Series, Prentice Hall PTR

Wilson D (2001) Teaching XP: a case study. XP/Agile Universe website

4 Time

Abstract

Time is addressed differently by different people and cultures; for example, in western culture, time is mainly associated with financial profit, i.e., "Time is money." Time plays a special role in software engineering: the project schedule should be met, the product should be delivered on time, teammates should complete their tasks on time, and so on. This chapter deals with how the time concept is expressed in software engineering in general and in agile software development in particular. In agile software development time is boxed for each activity, and when needed, instead of "moving" deadlines, the scope is changed according to customer priorities. This conception is supported by agile software development methods in different ways that not only enable one to work at a sustainable pace, but that also result in high quality products.

4.1 Overview

This chapter examines how time issues are expressed in agile software development environments in general; more specifically, it describes the planning activity that is part of the iteration Business Days, described in Chapter 3, Customers and Users.

In a specific iteration planning, the customer presents the relevant stories, and the team, based on a high level design prepared previously, carries out a planning session which includes work distribution, time estimation, and load balance

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_4, © Springer-Verlag London Limited 2008

among the team members. The goal of the planning session is to reach a complete yet sustainable plan for the implementation of the customer stories during the coming iteration. At the end of the planning session all teammates are familiar with the customer's vision and priorities and with the individuals' tasks for the coming iteration.

No change is introduced on the iteration development days. The customer can still change the project scope, the stories, and their priorities; still, these changes are implemented only in the next Business Days.

The details of the planning session are described in the section that deals with time in learning environments. The same process, with some adjustments, should be conducted in industry.

Additional topics discussed in this chapter include:

- time-related problems associated with software projects;
- a tightness model for the assessment of software development methods;
- the concept of working at a sustainable pace;
- time management with respect to software development methods.

4.2 Objectives

- Readers will get acquainted with agile planning activity, including time estimation of development tasks, work distribution among teammates, total time calculation for the iteration, and load balance among teammates.
- Readers will become aware of the time and pace notions with respect to agile software development.
- Readers will become familiar with how a week in the life of an agile team works.
- Readers will become familiar with time management in agile software development methods in general, and with measures as part of time management in particular.
- Readers will become familiar with a tightness model.

4.3 Study Questions

1. What is unique about the planning process of an agile software project?
2. What are the main characteristics of the agile planning sessions? Of the release? Of the iteration?

3. Describe methods for time estimation of software projects.
4. How do different software development methods relate to time management?
5. What measures can be used for time management in software projects?
6. What is the concept of team velocity? What is its importance? How do agile methods calculate team velocity?
7. What is the concept of sustainable pace? What is its importance?
8. Why, in your opinion, are many software projects characterized by schedule overrun? How would you suggest overcoming this problem?
9. Do you have personal successful experiences with respect to project schedules? If yes, explain the source of their success.
10. What are the similarities and differences between software projects and non-software projects? How do these similarities and differences relate to time issues?
11. One of the famous rules of software engineering is Brooks's assertion that "Adding manpower to a late software project makes it later" (Brooks 1975, 1995). Can you explain this rule? In your opinion, is it relevant for other professions as well? Why? In what ways does it highlight the importance of the time dimension of software projects?

4.4 Time-Related Problems in Software Projects

In his classic book *The Mythical Man-Month* (Brooks 1975, 1995), Brooks writes:

> More software projects have gone awry for lack of calendar time than for all other causes combined. Why is this cause of disaster so common?
>
> First, our techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption which is quite untrue, i.e., that all will go well.
>
> Second, our estimating techniques fallaciously confuse effort with progress, hiding the assumption that man and months are interchangeable.
>
> Third, because we are uncertain of our estimates, software managers often lack the courteous stubbornness of Antoine's chef.
>
> Fourth, schedule progress is poorly monitored. Techniques proven and routine in other engineering disciplines are considered radical innovations in software engineering.
>
> Fifth, when schedule slippage is recognized, the natural (and traditional) response is to add manpower. Like dousing a fire with gasoline, this makes matter worse, much worse. More fire requires more gasoline, and thus begins a regenerative cycle which ends in disaster (14).

Tasks

1. Explain each of Brooks's claims.
2. Do you have personal experiences that strengthen Brooks's claims?

It is clear that time occupies a crucial place in project management. It seems, however, that in software engineering time plays a special role. It also seems that time is one of the most important factors dominating software development. One reason that makes time so crucial in software development is that software development does not progress linearly. This, in fact, is expressed by Brooks's statement that, in software projects, months and people are not interchangeable (Brooks 1975, 1995).

4.4.1 List of Time-Related Problems of Software Projects

Based on Hazzan and Dubinsky (2007), we illustrate the significant role of time in software development by presenting time-related problems in the development process.


HOT

Bottlenecks. Bottlenecks in software development occur when one or more functions in the process await the output of another function in the process, with teammates having nothing to work on in the meantime. This may happen, for example, when quality assurance people wait for artefacts to work on or, vice versa, when developers wait for artefacts from quality assurance people. Another example is when developers wait for artefacts from system analysts, like the specification of a specific module.

Project planning and schedule. Two main problems are associated with schedules, which are, in fact, closely connected. The first is the mere construction of a *feasible* project schedule. The second problem is to meet the schedule that has been set.


HOT

Time estimation. There are different ways to support time estimation (Boehm 1981, Boehm et al. 2000, Kemerer 1987, SEI 2001). With respect to the estimation of the development time of a specific module/class/task, it is well known that the greater the module/class/task is, the more difficult it is to estimate its development time. Tomayko and Hazzan (2004) present evidence that the smaller the estimated unit is, the more accurate is its time estimation.


HOT

Time pressure. Time pressure is the result of the previous problems. It happens usually toward the end of the development process, when teammates cannot meet the project schedule, either because of poor time estimations or bottlenecks. Time pressure usually leads to the skipping of different testing activities, which in turn leads to a decrease in software quality. Van Vliet

(2000), for instance, says: "The testing activity often does not get the attention it deserves. By the time the software has been written, we are often pressed for time, which does not encourage thorough testing.... Postponing test activities for too long is one of the most severe mistakes often made in software development projects. This postponement makes testing a rather costly affair" (386–397).

Late delivery. Late deliveries occur as a result of inappropriate project planning, usually due to poor estimations. Data indicate that the percentage of software projects that fail to accomplish on-time delivery is quite high. See, for example, the data presented by the Standish Group Report (Standish 1994).

Tasks

1. For each of the above problems, specify how agile software development methods attempt to overcome it in general and, in particular, what agile practice(s) aims at solving it. If needed, go on reading this chapter before answering this question.
2. Analyze the above problems from the HOT—Human, Organizational, and Technological—perspectives.

4.4.2 Case Study 4.1. Software Organizational Survey from the Time Perspective

This case study illustrates some of the above mentioned problems that characterize software projects. The following data were gathered in a large company during an organizational survey conducted at the company in order to understand the roots of the problems the company encountered with respect to software development and to propose possible solutions to overcome those problems. We present here only data related to time.

Task

For each of the following data pieces, explain its source and suggest how agile software development attempts to solve it.

a. *One of the questions presented to developers was: Indicate two factors that you would improve in your team in order to develop higher quality software.*

   Out of 46 answers, 13 developers (28%) indicated the time element as a factor that should be addressed in order to improve software quality. Here are several illustrations:

- Planning a schedule that can be met.
- Allocation of time to learn the things [to be implemented] before we rush to the next coding; allocation of enough time for review and debugging.
- Allocation of time for design and education in design topics.
- Commitment of the software people to the schedule and the product.
- Add time to the development and testing [stages].
- Dedicate additional time to code review.

b. The developers were asked to describe the development process in the organization and to indicate the pros and cons of that process. Following are some of the time-related responses received.

Indicate at least three benefits of the development process that you have just described.

Out of 22 developers who answered this question, *none* mentioned the concept of time.

At the same time, time appears in the "pitfalls" question, as described below.

Indicate at least three pitfalls of the development process that you have just described.

Out of 22 developers who answered this question, 9 (41%) mentioned the concept of time, as is illustrated by the following quotes:

- It is difficult to estimate times (the scope is not really known).
- This process requires a lot of time and resources.
- It lasts too long.
- The schedule is tight and it requires making decisions that are detrimental to the quality of the development.
- Too many bad ideas are accepted because there is not enough time to establish a new process.
- The lack of parallel work leads to some inefficiency and redundancy in the development time; the requirements are not always well defined, a fact that takes a lot of the time of the software engineers and the system engineers during the DR [Design Review].
- The tests require too much time.

4.5 Tightness of Software Development Methods

This section shows that agile software development methods are tight, which means that, though they are flexible in terms of change introduction, they are very sensitive to time frameworks. This awareness of time frameworks is important since software is an intangible product, and accordingly its development should be kept strictly paced; otherwise, problems such as the ones described above emerge in many cases. Also, "tightness" suggests the idea that the tighter a development method is, the more ordered is the software development environment it inspires.

The tightness of agile methods is achieved in several ways. First, short iterations and releases are set. Second, agile methods set the time dimension fixed as well as the cost and quality axes, enabling changes only in the project scope (Beck 2000, Beck and Fowler 2000, Cohn 2006). The time setting is determined by the project timetable, as presented in Chapter 3, Customers and Users, by designing the project roadmap consisting of iterations of a fixed number of development days.

Schuh (2004) explains that "Agile projects perform time boxing at the release and the iteration levels, so a single agile project contains multiple timeboxes.... Time boxes force the customer to make decisions on the short-term direction of the project; second, they always provide a near-term goal, which can keep the entire team from wandering off target.... Finally, time boxes ensure that the team delivers something useful within a short and defined period.... The last two arguments closely refer to the first-things-first idea—if you have a four months release time you think you have a lot of time and you are not focused on what is important—to deliver working software to the customer" (154–155).

Beck and Andres (2005) analyze the importance of keeping the time axes fixed. They present the three variables—time, quality, and scope—as variables of which, in some software development processes, two are usually set and the third can vary. However, they claim, "This method doesn't work well in practice. Time and cost are generally set outside the project. That leaves quality as the only variable you can manipulate. Lowering the quality of your work doesn't eliminate work, it just shifts it later so delays are not clearly your responsibility. You can create the illusion of progress this way, but you pay in reduced satisfaction and damaged relationships. Satisfaction comes from doing quality work" (92).

Among other differences, software development methods vary with respect to their tightness level and the culture that the tightness level of the development method inspires. Connections between tightness and organizational changes are further examined in Chapter 12, Change.

In this book we adopt a relatively tighter approach, since we believe that the tightness level that characterizes a development method is one of the main factors that enable software projects to achieve their targets successfully.

Hazzan and Dubinsky (2005) conceptualize the tightness idea by defining the tightness level of a software development method along the following five dimensions, whereby the higher the values of these factors, the tighter the software development method.

- Project plan dimension: number of releases and feedback milestones, level of emphasis placed on planning, number of days for which specific planning is made (the smaller the number of days, the tighter the software development method).
- Procedures and standards dimension: level of detail that describes the software development method.
- Responsibility and accountability dimension: level of role performance, level of personal accountability, frequency at which team members are required to report on their progress.
- Time estimation dimension: importance given to time estimation, resolution level of time estimation (hours, days, months)—the smaller the time units, the higher the resolution, and the higher the value of this dimension.
- Individual need satisfaction dimension: mutual dependency of team members, level of planning that inspires the message "Invest now for the future."

Tasks

1. Analyze the agile practices presented in Chapter 1, Introduction to Agile Software Development, according to their tightness level.
2. For each dimension of the tightness model explain how it is expressed by the agile approach.
3. How does the tightness model inspire software project management?
4. In your opinion, what tightness level (low, high) of a software development method do teammates prefer? Why?
5. Choose two agile development methods and analyze them according to the tightness model.
6. Suppose we have a tool that measures the tightness level of a team in a similar way to the way in which we measure the tightness level of a development

method. Suggest possible scenarios for the adoption of agile development when the team tightness and the software development tightness fit each other and when there is a clash between the tightness level of the team and that of the software development method. Address questions such as: What would happen when the team tries to adopt the development method? What is the best situation for the application of an agile process?

4.6 Sustainable Pace

Tightness is one time-related characteristic of agile software development. Sustainable pace is another one. This section introduces the importance of the sustainable pace that agile methods inspire. Sustainable pace means that the development process is carried out in a reasonable number of hours, which are well planned and enable the team to be productive and to produce quality products.

The rationale for keeping a sustainable pace is that overworked programmers are unable to produce quality code. Since several agile principles, such as the whole team, pair programming, the planning session, and time estimation, ensure productivity during the development hours, agile programmers can work at a sustainable pace, be productive, and produce quality code.

Task

What characteristics of the agile planning session enable development at a sustainable pace?

The following data illustrate this productivity. Based on 31 Extreme Programming/Agile-methods early adopter projects, Reifer (2002) indicated a 25–50% reduction in time-to-market (188). These data are also supported by the results of the VersionOne and Agile Alliance survey conducted in 2007 (see Chapter 1, Introduction to Agile Software Development). This evidence shows that the agile approach inspires a productive and efficient working environment without working long hours. In other words, the agile approach shows that quality and productivity can be achieved at a sustainable pace as long as the work hours are managed efficiently.

As it turns out, the agile method is not the only one that advocates the sustainable pace concept. For example, in their software project management book, Hughes and Cotterell (2002) state: "There is good evidence that productivity and the quality of output goes down when more than about 40 hours a week are worked.... Clearly, it is sometimes necessary to put in extra effort to

overcome some temporary obstacle or to deal with an emergency, but if overtime working becomes a way of life then there will be longer-term problems" (226). Indeed, in some cases this principle is presented as a general guideline or as a recommendation. At the same time, however, in agile development environments this concept is one of the core principles of the approach; and for an agile team, working at a sustainable pace is an integral part of the development framework.

4.6.1 Case Study 4.2. An Iteration Timetable of an Agile Team

To illustrate the idea of sustainable pace, this case study presents an iteration timetable of an agile team. The team belongs to an organization which works in two-week iteration cycles. On the first day of the iteration the team runs the Business Day (see Chapter 3, Customers and Users). The nine remaining days of the iteration are development days. The team decided on the actual hours during which the team sits together in the collaborative workspace (see Chapter 1, Introduction to Agile Software Development) and carries out the development tasks. The decision was 10:00–12:30 and 13:30–16:00 (that is, five hours each day). At the beginning of each development day a stand-up meeting takes place. Each team member describes in two or three sentences what he or she has accomplished the day before and what he or she plans to perform for today. These brief reports address both the development tasks and the tasks associated with the personal role.

Following the development time slots, time is dedicated to professional development activities, project meetings, sub-project meetings and role-holder meetings (see Chapter 2, Teamwork).

Table 4.1 describes the schedule of a development day of that agile team. The notion of sustainable pace, reflected in nine work hours per day with five development hours, is clearly reflected in this timetable.

Table 4.1 A development day of an agile team

Hours	Activity
9:00–9:30	Personal arrangements: emails, phone calls, etc.
9:50–10:00	Stand-up meeting
10:00–12:30	Development
12:30–13:30	Lunch break
13:30–16:00	Development—Continuation
16:00–17:30	Other Tasks (e.g., project meetings, role and study tasks, role holders meetings)
17:30–18:00	Miscellaneous

Such a schedule is possible because of tight planning and the fact that when a team sits together and concentrates on the development task, time is exploited more efficiently. It can also be seen that each individual's time is shared between the accomplishment of the development tasks and the individual role.

Task

Discuss and build with your team a weekly sustainable time table. What considerations guided your discussion and decision making process?

4.7 Time Management of Agile Projects

This section reviews two ways in which time is managed effectively by agile methods. First, we review time related measures; second, we put the agile approach within Covey's time management framework of *First Things First* (Covey et al. 1994).

4.7.1 Time Measurements

One of the common measures with respect to software project time management is the time estimated for development tasks versus the actual time to develop them. In order to control the iteration progress, this kind of measure is inspected on a daily basis; in order to control the release progress, this kind of measure is also examined each iteration. The daily progress is measured against the iteration commitment conducted in the Business Day. The iteration progress is measured against the release commitment, shared at the beginning of the release.

The whiteboards of the collaborative workspace constantly present a graph in which the horizontal axis represents the iteration days and the vertical axis indicates the number of hours.

Each day, the tracker adds two new points to the graph that represent the project's progress. The first one—the "total expected" point—represents the cumulative estimations of all tasks that were *completed* up to the previous day; the second point—the "total done" point—represents the cumulative actual time devoted to those tasks. A completed task is counted only when the developer in charge completes its coding, unit testing, and integration into the developed system.

The following case study illustrates this practice.

4.7.1.1 Case Study 4.3. Measuring Estimations Versus Actual Development Time

Figure 4.1 presents three graphs: estimations, actual time in a specific iteration of a specific team, and the expected average pace according to available time (Dubinsky et al. 2008).

As can be observed, there is a significant difference between the allocated time for development (about 270 hours) and the time that was actually dedicated for development. Several factors may cause this gap: First, only completed tasks are presented; tasks that have been performed but have not been completed before the end of the iteration are not calculated, and the time that has been dedicated for their development is not reflected in the graph. Second, time invested in tasks that appear urgently, like support service to end users who work with deployed modules, is not presented in the graph. Third, there was a sudden absence of developers whose time was taken into account in the iteration.

Figure 4.2 shows another graph of estimations versus actual development time (Dubinsky et al. 2008). In this case, all the planned tasks were completed. Accordingly, the total-expected point unites with the expected-pace point, since this was the number of development hours considered in the planning session.

It can also been observed in Figure 4.2 that the time distribution among tasks was reasonably good. Further, the graph shape shows that the integration is continuous; too many dependencies between tasks would have delayed their completion and integration, reflected by a nearly flat total-done line until almost the end of the iteration, and then a sharp increase as many tasks are completed together.

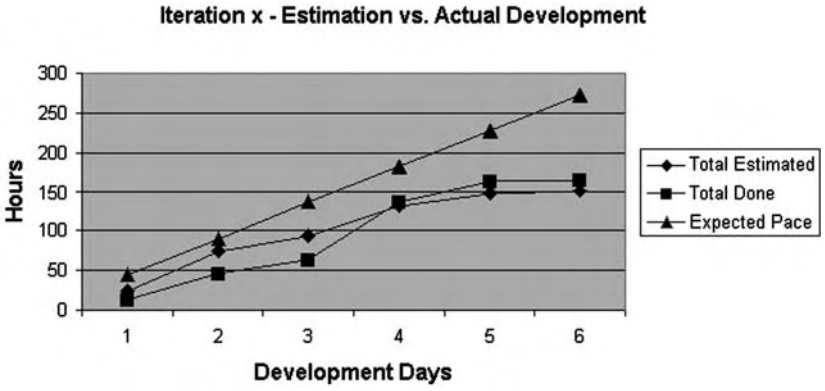


Figure 4.1 *Estimation versus actual development time: example 1.*

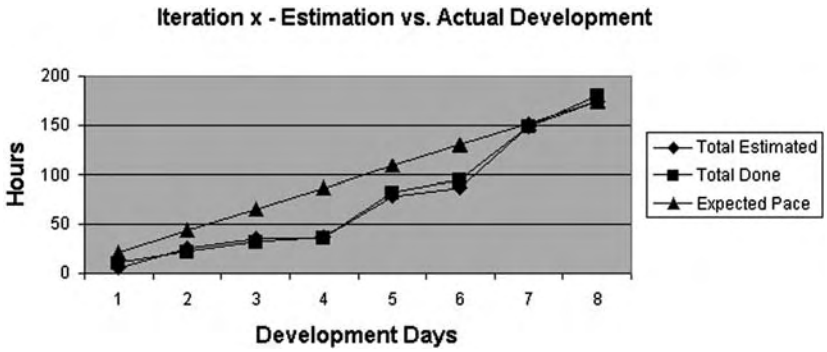


Figure 4.2 *Estimation vs. actual development time: example 2.*

Data such as those presented in Figures 4.1 and 4.2 is examined every day. Thus, if it seems that problems are expected with respect to the completion of the planned stories of the current iteration, the customer can reprioritize the stories and change the iteration scope. This way, stories that remain in the iteration scope can be developed and tested properly.

In addition, these data are examined when the iteration ends in order to learn about the team velocity. *Team velocity* can be perceived as the amount of productive work units per iteration (Beck and Fowler 2000). Measuring team velocity increases the visibility of the development progress and enables one to make decisions on how to continue with respect to functionalities and priorities. The data are examined also with all the other iterations completed so far in the release, in order to learn about the project's progress.

Task

How does such a measure support the project management?

4.7.2 Prioritizing Development Tasks

What is common to refactoring, test-driven development, and the planning activity, which are some of the basic practices of agile teams? Why are they important? After all, it can be argued that the only important activity is code production. The answer is that they all support the management of the software development process. This idea is further strengthened by the following case study that uses Covey's concept of *First Things First* (Covey et al. 1994). It

shows how agile software development helps agile teams focus on what is *important* rather than on what is urgent. This idea is manifested by different agile practices such as refactoring (always important but not urgent) and pair programming (helps the team stay focused and avoid the distractions of unimportant and non-urgent activities).

4.7.2.1 Case Study 4.4. First Things First

This case study illustrates how time management is manifested in agile software development environments and how agile software development increases the team members' awareness of time management ideas as they are manifested in agile software development environments.

This data set is taken from a team undergoing the transition to agile software development. The team is used to carrying out a one-hour reflection at the end of each iteration. The reflections that the team conducted at the end of the first two iterations were dedicated to learning processes (the first) and to prioritizing activities (the second).

A reflective session takes place during the Business Day, which, as has been described in Chapter 3, Customers and Users, also includes a demonstration of the features developed during the last iteration, a presentation of the measures taken, and the planning session for the next iteration. Additional details on reflective activities performed by agile teams are presented in Chapter 11, Reflection.

The rationale for the second reflection is based on the realization that time management is a key element of agile software development. Accordingly, its objective was first, to emphasize the importance of time management issues in software projects, and second, to help developers grasp how time management is expressed in agile software development.

More specifically, according to Covey's concept of *First Things First* (Covey et al. 1994), agile software development helps teams focus on what is *important* rather than on what is *urgent*. As mentioned above, this idea is manifested by different agile practices such as refactoring and pair programming.

Table 4.2 Reflection on time management

You are kindly requested to reflect on your role and personal work habits/processes in the project and to complete the following matrix accordingly:	
I. Urgent and Important	**II. Not Urgent but Important**
III. Urgent but Not Important	**IV. Not Urgent and Not Important**
After you complete the four quadrants, please formulate at least two rules/guidelines to apply when needed in order to focus on activities that belong in Quadrant II.	

In this reflective activity, the participants (team members, customer and management representatives) were asked first to work individually on the task presented in Table 4.2.

Tasks

1. Fill in Table 4.2 by analyzing your current software project development.
2. Describe three things that you learned from filling in the table (about yourself, about your work, about your time allocation, and about your team).

The aim of the above activity is to direct developers to focus on Quadrant II, which contains items that are non-urgent but important. As it turns out, these items are the ones we are more likely to neglect but should focus on in order to achieve effectiveness and quality. In the context of software development, this can be explained by the fact that people tend to be distracted from what is important because it is difficult to stay focused on the development of an intangible product, such as software. As mentioned above, and illustrated by the developers' responses presented in Table 4.3, agile software development guides developers to stay focused by implementing activities from the second quadrant—the *quality quadrant* (important but not urgent). Further, as this illustration indicates, this was very clear to the team after only two two-week iterations.

Table 4.3 A sample of developers' suggestions for each quadrant (© [2007] IEEE)

I. Urgent and Important	**II. Not Urgent but Important**
Production problems	Iteration planning
Fixing bugs that prevent progress	Design
Preparing a presentation after it has been postponed till the last minute	Learning new technologies
	Refactoring
	Tracking—follow-up and control
	Testing
	Taking care of infrastructure
	Preparing a presentation on time
	Taking care of procedures, target definition and responsibilities
III. Urgent but Not Important	**IV. Not Urgent and Not Important**
Working on management assignments that arrive late and have tight deadlines	Mingling
Helping other team members with urgent tasks that are not important to me	Personal arrangements/errands

Table 4.3 presents a sample of suggestions developers made for each quadrant. As can be seen below, Quadrant II—the quality quadrant—contains agile activities and practices.

During the discussion that followed the individual work, guidelines for items in the Important but Not Urgent quadrant, such as the following ones, were suggested:

- When there is a problem, "put it on the table" and talk about it (thus, we make it important).
- Allocate time for important but non-urgent issues.
- When the time is dedicated and we sit [to discuss the problem], do not waste time.
- Let as many people as possible obtain a wide perspective.
- Plan everything possible in advance, not to put things off till the last minute.
- As much as possible, do not perform tasks that are not connected to the current iteration.

The project manager noted: "*The second quadrant is characterized by teamwork—because of the team, I do what is important and I do not give up.*"

The above quotes illustrate that agile software development helps software developers focus on the second quadrant, inspiring a development process that is composed of important (but not urgent) activities and eliminating the performance of urgent activities (mainly the not important ones) during the course of the project.

4.8 Time in Learning Environments

4.8.1 The Planning Activity

The forth studio meeting aims at the completion of the planning session. The description fits for the industrial setting as well, with necessary adjustments.

Students estimate the duration of the different tasks. They are encouraged to give realistic estimates, rather than to succumb to the pressure to present overly optimistic estimations. The students are reminded that the time allocated for a specific task should include its unit testing as well (see Chapter 6, Quality).

We note that in some agile methods the personal distribution of tasks among teammates is not carried out during the planning session, but rather during the iteration itself.

Specifically, the development tasks are reviewed by the academic coach, who verifies that a realistic estimated development time is given for each task. Then he or she writes the different tasks on the board along with their estimated time. Time estimations are discussed with the students, especially if they are too short (less than two hours) or too long (over fifteen hours). The academic coach asks the students who suggested estimates exceeding fifteen hours to explain their estimations. Usually, it turns out that most of the time is needed in order to study the subject. In such cases, the students' attention is drawn to the fact that the long time estimation was due to this and that the coding and unit testing takes up only part of this time. In other cases, it happens either when a high level design has not been prepared or the task is too vague. In such cases, the breakdown into tasks is redone.

Next, the time estimations are added up, and the total available working hours the team has in the coming iteration is calculated as well. These two numbers are compared, and the comparison indicates whether customer stories should be added or some should be removed. In such cases, the customer prioritizes and adds or removes stories according to the situation in hand. Table 4.4 presents an example of such a time calculation.

Before the activity ends, the academic coach explains the practice of pair programming, and emphasizes its contribution to information sharing and collective ownership.

The planning session concludes with a final balancing of the development time among the students.

Table 4.4 Example of the calculation of the available development time for a team of twelve students

	Activity
Time available	Since the first presentation to the customer is done in the 7th week of the semester, and this meeting takes place at the 4th week of the semester, the calculation of the available development hours yields 360 hours and is described in what follows: 3 weeks until the iteration presentation (the 7th week of the semester) $\times$ 12 students $\times$ 10 hours per student per week
Pair programming factor	Let us assume that the total time estimation for all of the development tasks is 180 hours. Based on Cockburn and Williams (2000), the pair-programming factor is 1.15. Since the students are not yet familiar with this practice, a factor of 1.5 should be used. Accordingly, the 180 hours are multiplied by 1.5, and we get 270 development hours
Others	The remaining 90 hours are allocated for integration, presentation, and documentation purposes

4.8.2 Teaching and Learning Principles

The following two teaching and learning principles deal with the concept of time management. In our teaching and learning principles list, these principles are numbers 3 and 8. The complete list appears in Chapter 14, Delivery and Cyclicality.

4.8.2.1 Teaching and Learning Principle 3: Explain While Doing

This principle implies that when a concept, an activity, or a practice is first introduced to learners, the educator should *not* explain it too extensively, but rather should invite the learners to start applying it as soon as the basic relevant knowledge has been introduced. While the learners are performing the activity, the educator should then add the details gradually, refine the explanation, and reflect on the learners' activities.

A good illustration of the application of this principle is the teaching of how to carry out the planning session, through which the customer communicates to the development team his or her requirements, to be implemented in the next iteration, and the team members plan the actual development of the tasks allocated for the coming iteration. Since the planning session consists of many relatively simple stages and details, if learners are given all of these details before they start experiencing the planning session, then the global picture, as well as the logic of each stage, the order of the stages, and their interconnections might not be clear. Alternatively, if learners are first introduced only to the main ideas of the planning session and then start performing it, with the details gradually introduced during this process, the different stages will be situated better within the wider context and will serve to clarify the structure and logic of the entire process.

4.8.2.2 Teaching and Learning Principle 8: Manage Time

This principle relates to time management and is manifested in the course in different ways, mainly in the studio component, in which the students develop the software project.

First, students are required to arrive on time to all the weekly meetings.

Second, the students are committed to attend all meetings. If a student is forced to leave the lesson for some unexpected reason, it is an opportunity to discuss the analogy of such a case in real life situations.

Third, development at a sustainable pace is highlighted. Students are required to invest in the project development activities a specific and known number of hours, according to the course credit. The allocation of development tasks fits this number of hours and is determined in a transparent process during the planning session.

Forth, time estimation for the development tasks (and the assessment of this time estimation) is never skipped. Furthermore, learners are asked to estimate (and then to evaluate) the way their time is shared between the accomplishment of their personal development tasks and the performance of their personal roles (see Teaching and Learning Principle 7, explained in Chapter 2).

4.8.3 Students' Reflections on Time-Related Issues

Students' reflections on the planning activity show that the planning session supports their time management in general and increases their awareness of time-related issues in particular. Here are some students' reflections after the first and the third iterations of a specific project.

- "MUCH more time should be dedicated to the planning of the structure of the program, both interfaces and implementation, before anyone starts writing any code."
- "Things usually take more time than expected, especially because of integration and misconceptions—this must be taken into account when estimating times."
- "As for time evaluation, we met our estimations almost exactly, and in some cases even finished tasks sooner than we had expected."
- "In iterations 2 and 3 the times were better defined, because of the experience we had gained."

4.8.4 The Academic Coach's Perspective

The planning session gives the academic coach an excellent viewpoint on the students' work with respect to the project's planning, the design of its parts, and the development management. During the planning activity, the academic coach becomes extensively familiar with the project details and with each student's part in the development. This, in turn, enables a more precise evaluation of each student (see the evaluation scheme presented in Chapter 2, Teamwork).

At the last iteration, the coach may add a change request to the project in order to illustrate the fact that new customer change requests can be added at any time, provided they fit into the project time framework. In such cases, the student whose role is the customer (if no real external customer participates) is asked to omit one or more story to enable the addition of the new change request.

In general, the planning session provides a coaching framework from a time management perspective. Following are some remarks of academic coaches with respect to this activity:

- "I now have a much better feeling about how to lead a project. Leading a project by giving out the work and simply waiting for them [the students] to come to you with questions or results is one way. It's totally different, however, when you have a group, and you lead it through the project in one way or another. It gave me experience managing a project. Experience in different aspects: one is time scheduling, another is the planning of the work, and another is the professional side of programming."
- "As for the *planning session,* I really saw that it was important; they worked with the cards and wrote their contents into the computer."

4.9 Summary and Reflective Questions

1. Select one of your development requirements/stories, and perform the following activities with respect to it:
   a. prepare a high level design;
   b. break it down into development tasks;
   c. estimate the development time for each task, including development and unit testing;
   d. calculate the total time—that is, sum up the estimated development time for all the tasks;
   e. develop all tasks and measure the actual time invested;
   f. sketch a graph of estimations versus actual development times;
   g. discuss lessons learned.
2. Select two time-related issues presented in this chapter that you find interesting. Explain why you selected them and describe how the agile approach deals with these issues.

3. Indicate two time-related characteristics of software development methods. Discuss their expression by agile development methods.
4. Analyze the concept of time as it is manifested in agile software development from the HOT—Human, Organizational, and Technological—perspectives.

4.10 Summary

This chapter discusses the concepts of time and time management and presents the planning activities carried out in agile software development environments. The way the agile approach refers to time is "tight"—the main activities carried out in agile development environments are time boxed and are measured with respect to their actual development time in hour resolution. These characteristics ensure a controlled development process that enables the developers to increase the product quality. The concept of sustainable pace is also explained and illustrated in this chapter.

References

Beck K (2000) Extreme programming explained: embrace change, 1st ed. Addison-Wesley, Reading, MA

Beck K, Fowler M (2000) Planning extreme programming. Addison-Wesley, Reading, MA

Beck K, Andres C (2005) Extreme programming explained: embrace change, 2nd ed. Addison-Wesley, Reading, MA

Boehm B (1981) Software engineering economics. Prentice-Hall, Englewood Cliffs, NJ

Boehm B, Horowitz E, Madachy R, Reifer D, Clark BK, Steece B, Brown AW, Chulani S, Abts C (2000) Software cost estimation with COCOMO II. Prentice-Hall, Englewood Cliffs, NJ

Brooks FP (1975, 1995) The mythical man-month—essays on software engineering. Addison-Wesley, Reading, MA

Cockburn A, Williams L (2000) The costs and benefits of pair programming. 1st international conference on extreme programming and agile processes in software engineering. Italy

Cohn M (2006) Agile estimating and planning. RC Martin Series, Prentice Hall PTR, Englewood Cliffs

Covey S, Merrill AR, Merrill RR (1994) First things first. Free Press, New York

Dubinsky Y, Yaeli A, Feldman Y, Zarpas E, Nechushtai G (2008) Governance of Software Development: The Transition of Agile Scenario, IT Governance and Service Management Frameworks and Adaptations, Idea Group Publishing, Information Science Publishing, IRM press

Hazzan O, Dubinsky Y (2005) Clashes between culture and software development methods: the case of the Israeli hi-tech industry and extreme programming. Proceedings of the agile conference, IEEE computer society, Denver, CO, pp 59–69

Hazzan O, Dubinsky Y (2007) The software engineering timeline: a time management perspective. Proceedings of the IEEE international conference on software—science, technology & engineering, Herzelia, Israel, pp 95–103

Hughes B, Cotterell M (2002) Software project management. McGraw-Hill, New York
Kemerer CF (1987) An empirical validation of software cost estimation models. Communications of the ACM
Reifer DJ (2002) How to get the most out of XP/agile methods. Proceedings of the second XP universe and first agile universe conference, Chicago, IL, pp 185–196
Schuh P (2004) Integrating agile development in the real world. Charles River Media
SEI (2001) Standard CMMI appraisal method for process improvement (SCAMPISM). Software Engineering Institute
Standish group chaos report (1994) http://www.standishgroup.com/sample_research/chaos_1994_2.php
Tomayko J, Hazzan O (2004) Human aspects of software engineering. Charles River Media
Van Vliet H (2000) Software engineering—principles and practices. Wiley

5
Measures

Abstract

There is a consensus that the control and management of any process and activity can be improved by using measures to monitor its performance. In general, software engineering is a discipline that acknowledges products measures as well as measures of the development process. Specifically, the agile approach promotes a constant tracking during the entire process of software development. For example, the team velocity explained in Chapter 4, Time, is one of the important ways to control the project's progress. Further, the tracker role is responsible for the definition and refinement of the team measures, for the collection of the required data, and for the measure presentation. Some measures are presented daily, like the daily progress within the iteration; some measures are presented every iteration, like code coverage or iteration progress within the release; yet other measures are presented every release, like customer satisfaction or level of product testability. By using measures on a regular basis, all teammates and stakeholders can view them, give feedback, and suggest refinements, and thus become accountable for the project development.

5.1 Overview

This chapter focuses on how agile teams use measures—an important practice that agile teams apply to increase project transparency and reduce cognitive complexity. The idea is to help the agile team sense the development process of

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_5, © Springer-Verlag London Limited 2008

an intangible product—software. The need for such a practice is clear if we compare the development process of software to the development process of a tangible product, such as a road. In the road case, a professional person can estimate how long it will take to pave a specific part of the road based on previous experience and on measurements of how many miles have already been paved for how many hours or days. In software development it does not work so easily.

First, we cannot just look at or measure what portion of the software has been developed so far. Second, it is not clear how the measures should be taken—that is, according to what parameters the progress obtained so far should be measured: shall we use number of lines of code? No, it is well known that that is not a good indicator. Shall we measure progress by the number of work hours? Maybe, but still it is well known that software development does not progress linearly. Shall we use the number of tests we developed today, or during the last week? This might be a good measure; it will be discussed later in this chapter. What additional measures should be taken?

These measures, however, unlike those of the road, will not tell us exactly how far we are in the development process and what we still have ahead. This is because the ultimate target of software projects is not always clear at the early stages of development. For example, new requirements may emerge as the customer improves his or her understanding of the developed application; the team members may find out that the previous design they used does not capture the new requirements; etc. Therefore another kind of measure is needed for software development. This topic is the focus of this chapter.

In order to get at the chapter's main ideas, we first answer the following questions as they are applied in agile software development environments:

- Why are measures needed?
- Who decides what is measured?
- What should be measured?
- When are the measures taken?
- How are the measures taken?
- Who takes the measures?
- How are the measures used?

To illustrate the answers to these questions, two case studies are presented with specific examples of measures and an explanation of how the measures helped the engineers monitor the development process.

5.2 Objectives

- Readers will understand the importance of measures in software development.
- Readers will understand what measures are needed for monitoring software projects.
- Readers will experience the formulation of measures for their software project.
- Readers will learn the tracker role and the responsibilities related to this role.

5.3 Study Questions

1. How did you control the development process of your last software project?
2. Suggest measures that would enable you to improve your control of the development process? For each measure explain in what ways it might support the development process.
3. How would you suggest collecting data in practice? With respect to two specific measures, refer to the frequency of the measure collection, their display, and any additional factor that seems to be relevant for this purpose.

5.4 Why Are Measures Needed?

Measures are used in order to control and monitor software development processes and products. There are software projects in which measures are either not taken at all, or if taken, do not explicitly serve specific goals.

Accordingly, a set of measures, defined for a specific software project, should adhere to the following characteristics:

- The measures should be mapped to the project goals. It is recommended that this mapping be regularly assessed in order first, to ensure that no redundant measures are taken; and second, to check compliance with the different goals based on the existing measures (Dubinsky et al. 2008).
- The measure collection should not affect the process performance or product that the measures control.

In the case of a specific software project, a goal might be to shorten software delivery time. Consequently, among different sub-goals related to this goal, one can monitor on a daily basis the progress of a specific software project. One specific measure can be the rate of new check-in operations, performed each day, of an agile iteration. In order to learn about the progress of a specific project that is part of a given organization, this data is viewed on a daily basis, an iteration basis, and a release basis.

Measurements enable an agile software development team to get constant feedback from the different components of the software development environments, people, and code. A measure which is people-oriented might be, for example, customer satisfaction, or the amount of team overtime hours (for sustainable pace measure). A measure which is code-oriented might be code coverage, which is the level of coverage that unit tests give, or the number of defects. The ongoing presentation of the measures increases the project's transparency. Further, since agile software developments are open to (see the Agile Manifesto in Chapter 1), it is possible, if necessary, to replace a set of measures during the course of the project development process, or to decide on different measure sets for different projects within the same organization. One general rule, however, which is derived directly from the Agile Manifesto, should be followed: Measures should support and assist the individuals involved in the software development process.

Tasks

1. Explain how measures increase project transparency. Use specific examples of measures to illustrate your explanations.
2. Describe a team goal, measures that can help the team to achieve its goal, and how the measures can guide the team's decision making process.
3. Describe an organizational goal, measures that can help the organization to achieve its goal, and how the measures can guide the organizational decision making process.

5.5 Who Decides What Is Measured?

In an agile process, measures are determined by the customer, the development team, and the organization; each party decides what to measure based on its interests in the development process and artefacts.

The customer is interested in measuring the development progress and the quality of the artefact's performances and stability; the development team is interested in measuring the impacts of the development methodology, the satisfaction of the people involved, and the quality of the artefacts from technical perspectives, such as maintainability and extensibility; management people are interested in business aspects, like project costs and return on investment, as well as customer satisfaction.

Task

Give an example for at least one measure that you would define, if you were a customer, a team member, or a management member. For each measure define its purpose and how it would help you improve performance. Describe specific scenarios in which these measures would be helpful.

5.6 What Should Be Measured?

We measure artefacts that answer specific questions derived from specific goals. For example, suppose the team goal is to increase their productivity. Some questions that can be derived from this goal are: how many hours per day do teammates work to produce code? How many hours per day do teammates work in pairs? What is the pair's turnover? What is the actual size of the development work? The measure set that helps answer such questions should fit the situation and the involved individuals. In another case, for example, teammates need to decide about the best way to measure the size of their project, and which information should be gathered from each pair. The set of measures should be refined and adapted when needed. In the first case, for example, teammates can measure the number of hours they work together in the collaborative workspace, as another indicator for the productivity measure.

Measures should be as simple as possible to enable their actual measurement, as well as interpretation, by the different players participating in the development process. For example, if teammates are requested to report every fifteen minutes about their position in pairs, it will become annoying; instead, it is simpler to check a box in the team's planning tool every time a pair is changed.

Only several measures should be chosen; a large set of measures can negatively influence the development itself, since many hours will be needed for the measurement process. Hence, a reasonable and refined set of measures should be used. This number of measures, however, should fit the team's, the customers', and the organization's needs. Our rule of thumb is that the tracker should invest no

more than 20% of his or her time for the collection, presentation, and refinement of the set of measures.

Tasks

1. Suggest a situation in which too many measures negatively influence the software development process.
2. What principles of the Agile Manifesto are supported by the measures taken for a given software project?

5.7 When Are Measures Taken?

Agile software development requires constant feedback. Therefore, there are activities, like continuous integration, that should be measured several times each day. In such cases it is preferable that measures be taken automatically. Other activities can be measured on a daily basis. For example, measures that reflect the number of hours invested each day in development tasks and the hour distribution among the tasks completed during the day, can be taken on a daily basis. Measurements on a daily—or iteration—basis allow reflection (see Chapter 11, Reflection) on the progress of the process as often as possible.

Tasks

1. Suggest additional examples for measures that should be taken several times a day, on a daily basis, and on the iteration basis. Specify the purpose of each measure.
2. What principles of the Agile Manifesto are supported by the frequency of measure collection?

5.8 How Are Measures Taken?

Though there is a set of agreed upon measures, in order to foster and support measure collection, they are taken by the different role representatives assigned within the agile team (See Chapter 2, Teamwork).

The tracker is responsible for the measure collections and their presentations. One basic measure compares the developer's time estimation of the development tasks with the actual development time (see Chapter 4, Time).

The tester is responsible, among his or her other activities, for devising measures that deal with the code quality.

The user evaluator is responsible for measures that reflect the users' involvement in the design. And so on.

Task

Choose two roles. For each role suggest two measures that can support their accomplishment.

5.9 Who Takes the Measures?

All team members are involved in measuring the software development progress, either by reporting essential information to the team members who are responsible for specific measures or by measure gathering, analysis, and presentation.

5.10 How Are Measures Used?

It is not sufficient to observe the measures on a regular basis in order to communicate the project status among the different individuals and stakeholders involved in the development process. In addition to measure examination on a regular basis, after each period, preferably after each release that longs about two month (see Chapter 3, Customers and Users), the information derived from the measures and their analysis, should be evaluated against the project's goals. During such an iterative process, the goals and their sub-goals are refined, the measures are determined and changed if needed, and the actual information is assessed to check its compliance with the current goals.

In the agile spirit, the conclusions derived from such an examination are communicated to the team members, the customer(s), and the management, whether by actual participation in the examination sessions or by other means found appropriate for a specific project setting.

Tasks

1. Assume that not all the project stakeholders can participate in measure examination meetings (for example, the team is dispersed). Suggest other means by which the results of such an examination can be communicated to those who did not participate in the examination meeting itself.

2. Analyze the main ideas presented so far with respect to measures within the HOT—Human, Organizational, and Technological—analysis framework.

5.11 Case Study 5.1. Monitoring a Large-Scale Project by Measures

This case study illustrates how measures are used for a large-scale project, that the implementation of the agile approach for its development process was considered a risk (Talby et al. 2006). However, the agile approach was chosen to be applied for its development based on the analysis of the problems which the projects faced and the realization that the agile approach might help with these problems.

Accordingly, the project began with close management supervision and only one team. This implementation started with ambivalent feelings. On the one hand, it started with the hope that this team would be a prototype for the implementation of the agile approach among the other teams of the project; on the other hand, the beginning was accompanied by fears and concerns expressed about the possible incompatibility of the agile approach with the environment in which the project was developed. Consequently, it was determined to constantly evaluate the development process as part of the tracker role.

In what follows we first describe the measure definition and then illustrate the actual measures.

5.11.1 Measure Definition

The measure design begins with a project risk analysis. During this analysis, a measure is added when it seems valuable for risk reduction purposes.

In what follows, the four measures used in this project are described, together with the kinds of data gathered for their calculation. These four measures reflect the project status, presenting information about the total work and quality of work performed so far (*product size* and *faults,* respectively*),* the work progress pace (*pulse*), and the status of the remaining work versus the remaining human resources (*burn down*).

Product size is the first measure. It aims at presenting the amount of completed work. The data selected to reflect the amount of the completed work is the number of test points. One test point is defined either as one test step in an automatic acceptance test scenario or as one line of unit tests. The total number of test points that pass successfully is calculated for each kind of test (either acceptance test or unit test) and is gathered per iteration per component.

Additional information is gathered with respect to the number of test points for tests that pass, the number of test points for tests that fail, and the number of test points for tests that do not run at all.

This product size measure is very effective in delivering the following message: test points are the *only* measure that reflects the project productivity—nothing else (like the number of lines of the running code, for example) counts.

The product size measure is designed to cope with the risk related to the inability to measure the project's progress before the agile approach has been applied, and, consequently, the inability to compare its current velocity to that of the organization's previous development process (Beck and Fowler 2000, Cohn 2006). The advantage of test points over, for example, the number of lines of code or lines of specifications, is that the number of test points for a given feature is usually proportional to the feature's size and complexity. This is not the case with respect to the number of lines of code or lines of specifications.

Another significant risk that this measure deals with is the not thoroughly tested product; that is, a product that is not totally and systematically covered by tests. Such a situation arises when developers do not write or run enough tests. This observation, in fact, is not surprising. It is known that developers do not like and resist testing (Hazzan and Leron 2006). In the case of the project described in this case study, many of the team members had had previous experience in other projects in the organization, in which the artefacts which they developed were tested after the development phase—for example, by the quality assurance (QA) team.

Pulse is the second measure, which aims to measure how continuous the integration is. The data are automatically gathered from the development environment by counting how many check-in operations occur per day. Data are gathered for code (that is, code plus its unit test) check-ins, automatic-acceptance test check-ins, and detailed specifications check-ins.

The risk that the pulse measure is designed to monitor is high overhead due to lack of continuous integration. Agile software development requires a different mindset than the one the developers in this project were used to: instead of completing a two-week specifications task and only then starting the development phase, with the agile approach an entire iteration is set to be two weeks long, during which a full cycle of specification-coding-testing is completed, and usually more than one cycle per each teammate. When keeping a daily pulse, i.e., ongoing check-in operation of tested code, the overhead on integration and bug fixing is reduced.

Hence, the preliminary role of the pulse measure is to verify that integration is spread evenly across iterations. Accordingly, *steady* means that the pulse is more or less equal across many days and reflects a desired status; *spiky* means that most of the check-ins are grouped at the end of iterations, which in turn means that the developers do not integrate enough during the iterations. Naturally, spiky pulse reflects a negative signal.

Burn-down is the third measure. It presents the project's remaining work versus the remaining human resources. This measure is supported by two tables.

The first table is the main planning table that is updated for each task according to its kind (code, a test, or a detailed specification); opening and closing dates; estimated and real development time; and the component it belongs to.

The second table is the human resources table that is updated when new information about a teammate's absence is received. This table also includes the information about the allocation of a product's component to each teammate, with the percentages of her or his position portion in the project.

The data presented in these tables organize the *burn-down* chart by the remaining work in days versus the remaining human resources in days. This information can be presented per week or for any number of weeks till the release is completed.

The burn-down graph answers a very basic and important managerial question: are we going to meet the release goal, and if not, what can we do about it? Thus, it enables us to cope with the risk of missing the release goals due to the lack of a clear view of the project's progress and the release availability during the release.

The burn-down is useful both on the team level and the organization level. Based on the information obtained from the burn-down chart, a team leader, for example, can change the teammates' task allocation during the release according to high-priority components; upper management can easily verify whether the release is on track.

Release goals are set before each release—each goal is defined by a high-level feature, specified by the customer. Their accomplishment is verified by comparing a rough estimation of the effort required to complete each goal (given by the development team) with the total available resources. Once goals are defined and estimated, the calculations of the remaining work and the remaining resources are based on this initial estimation, which is refined as the release progresses.

Faults is the fourth measure, which counts faults per iteration. During the release, all faults that were discovered in a specific iteration were fixed at the beginning of the next iteration. The faults measure is required to continuously measure the product quality. It is important to note that the product size measure does not reflect the product quality, since although it measures test points, it does not correlate the number of failed or un-run test steps to the number of bugs which are currently open.

5.11.2 Measure Illustration

The presented data were gathered throughout the first release of the project, which consisted of four two-week iterations. The measures were presented during the iteration summary meetings, every two weeks before the planning session

(see Chapter 3, Customers and Users), and were also continually available on the project's Intranet. Most of the participants in the planning session reported that they were viewing the measure status only when it was presented in the iteration summary meeting every two weeks. The project leader, however, stated that the measures caused a change in the teammates' behavior, especially with respect to the writing of more tests.

Product Size. Usually, we do not view software product size through its tests. However, this is one of the interesting aspects of the product size measure. Figure 5.1 shows a global view of the four iterations, reflecting the growing numbers of test points as the product is developed. The significant growth in the last iteration is explained by the relatively small number of testers' hours for automatic test writing that were allocated to the project at first, and that soon became a bottleneck. In the third iteration, for example, not all coded features were tested, and accordingly the size measure showed only a small increase. Consequently, it was decided that at the beginning of the fourth iteration the main tester would teach the developers to write automatic test scenarios for their code. Accordingly, during the fourth iteration she wrote fewer tests by herself. The result was a sharp increase in product size during the forth iteration.

There is an option to drill down into the data in order to observe the product size per component. Figure 5.2 presents the number of test points per component per iteration. As can be observed, the fourth iteration was the most fruitful. A new component was dealt with, while the number of test points for the other

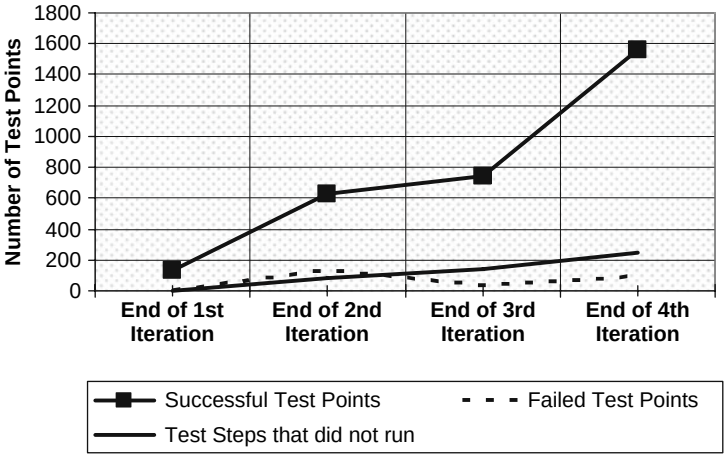


Figure 5.1 *Size measure during the release (© [2006] IEEE).*

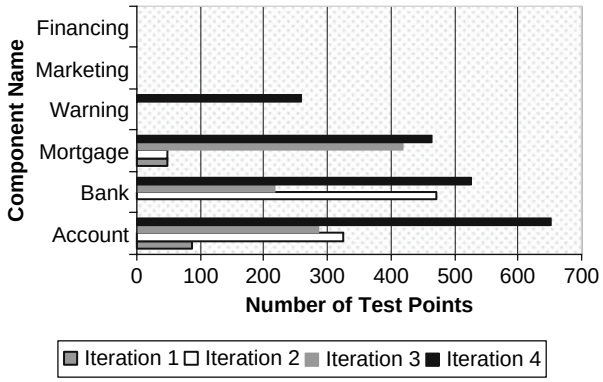


Figure 5.2 *Size measure according to components (© [2006] IEEE).*

components increased as well. This reflects a more mature stage of the development process.

Pulse. When the pulse measure was first presented to the team, some slight resistance to it was expressed. Several teammates argued that this measure does not really reflect continuous integration. One team member explicitly said that this measure searches for ratings like web surveys, and, accordingly, he suspected that teammates would simply click for check-in operation, just to raise the count. Other team members admitted to understanding why someone would do so. All signs of resistance disappeared when the teammates realized that first, only real check-in operations are counted—meaning a check-in is counted only when there is a change in the integrated part; and second, that this measure reflects more than continuous integration—that is, it actually indicates continuous work. Figure 5.3 shows the pulse measure for the entire release.

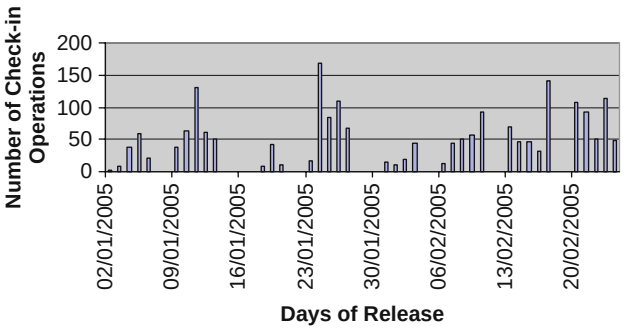


Figure 5.3 *Pulse measure during the release (© [2006] IEEE).*

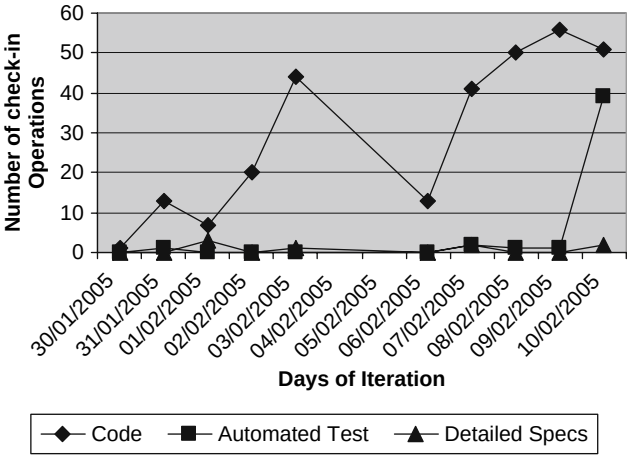


Figure 5.4 *Pulse measure of iteration 3 (© [2006] IEEE).*

As can be observed, the first week of each two-week iteration has fewer check-in operations than the second week of the iteration. Also, in the forth iteration the work has been distributed in the best way among the iteration days.

Figure 5.4 presents daily information about the pulse measure of the third iteration with the details of the different kinds of check-in operations. Note that check-ins of code files are the most prevalent, because code tends to be spread over more files than specifications of test scenarios, which in the case of this project were acceptance tests.

Burn-down. The burn-down measure is a classical managerial measure that shows whether the project plan can actually be accomplished. Figure 5.5 presents the burn-down measure for the entire release, derived from the estimations of the release and the total resource allocation for the entire release. Each pair of points represents data that were known at the beginning of the specified week. At the beginning of the first week, before the release started, the resources were 387 days for the entire release, while the work was estimated at 370 days. During the release, the number of resource days, as well as the number of days estimated for the development of the remaining work, were reduced. The data for the last (the eighth) week show the eighth week itself, where the number of days as resources is 49 and the estimated work is 46.25 days.

This kind of burn-down measure gives a two-month view of the development process and can serve as a successful plan chart.

Figures 5.6 and 5.7 present drill down data of the fifth and sixth weeks that are part of the third iteration. These figures present the inner data of the burn-down

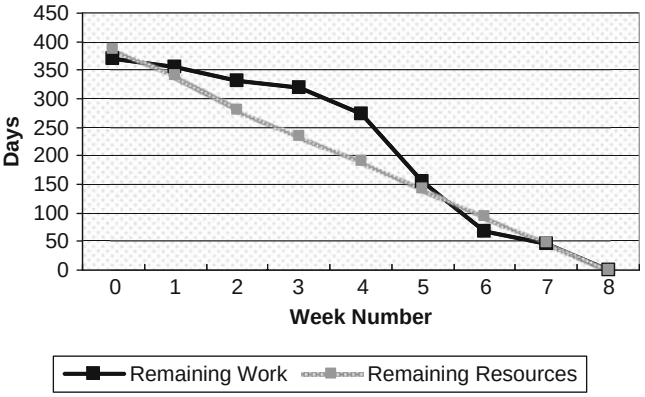


Figure 5.5 *Burn-down measure during the release (© [2006] IEEE).*

measure that show the remaining human resource days versus the remaining work days for each of the product's components.

In addition to the product's components, overhead was also referred to. Overhead includes training sessions, the Business Days, coordination meetings, and other activities which are not mere development activities. People were not allocated specifically for the "overhead" component—this is why its remaining resources are zero during the entire release, and it is supposed to be spread relatively equally across all team members. The simplest and most effective way to achieve this load balance was to require a small positive gap between resources and work in each component, which is reserved for the shared overheads. Note that the accumulated burn-down chart (shown in Figure 5.5) does contain the gap (remaining resources are greater than remaining work at the start).

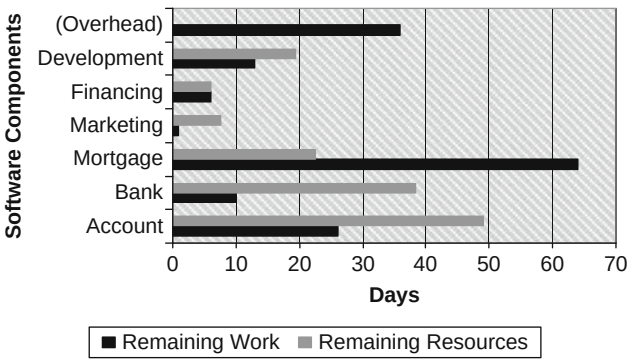


Figure 5.6 *Burn-down measure at week 5 (© [2006] IEEE).*

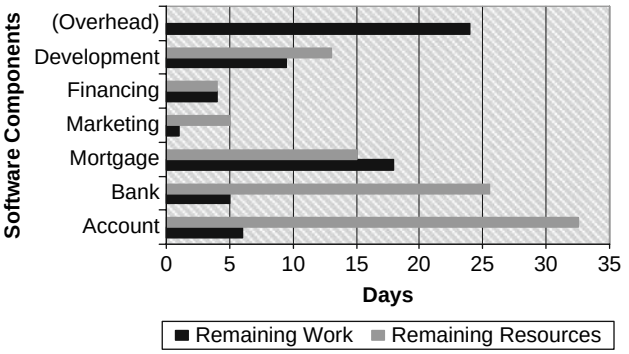


Figure 5.7 *Burn-down measure at week 6 (© [2006] IEEE).*

As can be observed, the variation among components is significant. This illustrates the strength of this measure with respect to the ability to decide about human resources mobility. For example, in Figure 5.6 the "Mortgage" component was below its goal in week 5. When this was observed, it was fixed in week 6, by adding human resources and reducing the number of features allocated for the release. Note also that the problem was not visible by looking at the accumulated burn-down chart alone, since the "Bank" and "Account" components have surplus human resources in these weeks that hide the lack of human resources for the "Mortgage" component.

Faults. The faults measure is a standard quality measure that indicates the number of faults found and what kind they are. They could be coding errors or detailed specifications errors. Figure 5.8 presents the number of faults per iteration, along with their distribution. Note how spec errors are common at the

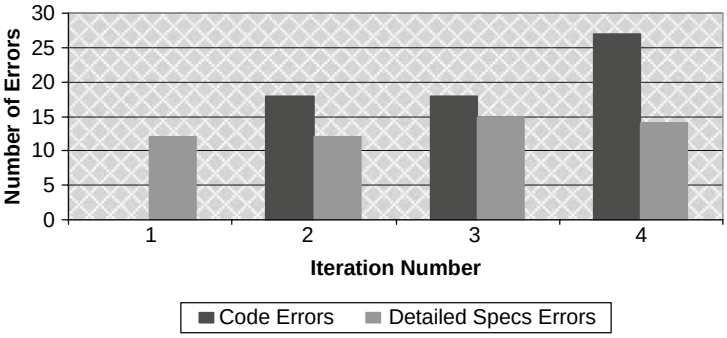


Figure 5.8 *Fault measure during the release (© [2006] IEEE).*

beginning of the project, when many team members are inexperienced at using the agile method, but slowly reduce their *relative* rate.

Tasks

1. How are the different agile characteristics and elements with which you are familiar at this stage of the book expressed in Case Study 5.1?
2. What additional lessons did you observe while reading Case Study 5.1?

5.12 Measures in Learning Environments

5.12.1 Teaching and Learning Principles

We present one teaching and learning principle that deals with how a development method is related to the real world of software development. In our list of teaching and learning principles, this is number 11. The complete list appears in Chapter 14, Delivery and Cyclicality.

5.12.1.1 Teaching and Learning Principle 11: Emphasize the Software Development Approach in the Context of the World of Software Development

Since the world of software engineering has witnessed relatively many cases in which new terms emerged and soon turned out to be no more than buzzwords, when teaching a software development approach, it would be preferable to connect that approach to the world of software development in general, and to other software development approaches in particular.

This can be done, for example, by presenting specific problems faced by the software industry, and illustrating how the particular approach taught—in our case, agile software development—can help overcome them. Learners will then feel, on the one hand, that they are being introduced to a development approach that is part of the software industry world and is not merely a passing fashion; and on the other hand, that the approach in question has emerged as a timely answer to the needs of the software industry, and that it will be useful to them in the future.

In the case of teaching the agile approach, the need for agility in software development may first be explained, and some problems involving other software development processes may be outlined. Such a perspective enables learners to

understand the role of the agile approach in the software industry and to observe that software engineering is still a developing field. This ability to cope with changes may be also required in the future, when other approaches to software development are formulated to meet the future needs of the software industry, which are currently still unknown.

This principle suits this chapter, since many software teams in the industry do not use measures on a regular basis. Further, goals are not set clearly and measures are not defined to answer whether we comply with the goals or not. The agile approach, that encourages measures as presented in this chapter, is one suitable way to close this gap.

5.12.2 Measurement Activities

In this studio meeting, students

- define the measures,
- decide on how to collect the relevant data,
- analyze and present the measures,
- assess the project status by using the measures.

In what follows we present a set of activities to perform related to measures.

5.12.2.1 Activity 1: Measure Definition

- Individual work: Write down two goals that in your opinion are the most important for the project's success.
- Open discussion about the project goals. At the end, two main goals are determined and are posted on the whiteboard.
- Group work of three or four students: Break down the two main goals into measurable sub-goals.
- Open discussion about the nature of measurable sub-goals.
- Group work: Exchange sub-goals among groups, and for each sub-goal received suggest appropriate measures.
- Open discussion about the project measures: The agreed upon measures are posted on the whiteboard, including their titles, a short description, their relation to the main project goals and sub-goals, ways to collect data for their measurement, and the teammates who are in charge of their measurement.

- Individual reflection: Write down two things that you learned about your project during this activity.

5.12.2.2 Activity 2: Data Collection

The team members who are responsible for the measures either explain to the other teammates what data they expect to report on when the measure is manually gathered, or inform the team about the automatic way by which information is gathered.

Sometimes, teammates do not cooperate with the team members who are responsible for the measures and do not report the requested data. In this case, an activity about cooperation in software teams is facilitated (for an example of such an activity, see the bonus allocation activity described in Chapter 2, Teamwork).

5.12.2.3 Activity 3: Measure Analysis and Presentation

The measure analysis and presentation are usually carried out by the tracker and the other team members who are responsible for specific measures. These role-holders:

- present the *weekly* measures in each stand-up meeting;
- present the *iteration* measures in each iteration planning;
- present the *release* measures at the end of the release (in the case of academia, a semester).

In industry, when the team meets daily, teammates and customers can decide on measures that will be presented on a daily basis, such as the amount of work accomplished each day, the number of tests written and passed successfully each day, as well as other measures according to the team's and the customers' needs.

5.12.2.4 Activity 4: Assessment

In each iteration planning, the following activities are performed:

- Individual reflection: Write down two things you learned from the measure presentation.
- Group work: Check the measure's compliance against the project's goals.

- Open discussion about the measure's compliance with the project's goals.
- Group work: Check if there is a need to refine/change the measures.
- Open discussion about the need to refine/change the measures.

5.12.3 Case Study 5.2. Role-Related Measures

This case study focuses on an agile project developed in the university by a team of ten students and illustrates how the role scheme described in Chapter 2, Teamwork, enabled the definition of measures that assisted in viewing the project's progress.

5.12.3.1 Defining the Measures

The role scheme enables us to define three main measures. The first is the *role time* measure. It measures the ratio of development to role performance; that is, the time invested in development tasks relative to the time invested in role activities. The second measure is the *role communication* measure that indicates the level of communication in the team at each development stage. This measure evolves because each role-holder needs to communicate with the other team members in order to perform his or her individual role efficiently. The third measure is the *role management* measure that indicates the level of project management. Since the role scheme aims to cover all management aspects of the project, the maximal level is obtained when all role-holders provide the maximum role performance.

These three measures, derived from the teammates' input on a weekly basis, as well as by an examination of the students' electronic forums and their personal role reports, can be viewed as ongoing indicators of student interaction. Although the agile approach emphasizes face-to-face communication among team members, when it is applied in a university environment several adaptations should be made. One adaptation, for example, that is relevant for this case study is the addition of electronic communication via an electronic forum, during the weekdays between the face-to-face compulsory sessions.

5.12.3.2 Illustrating the Measures

Role Time. Data related to the role time measure started being collected even before the first planning session took place. At this stage most of the students predicted a role time ratio of 70% development to 30% role, and only a few predicted 60 to 40% or 80 to 20%. Students talked about the role time variation

that is expected with respect to some of the roles in their role-peak weeks relative to their role-non-peak weeks. During the semester the students reported their own role time ratio only a few times. The semester average was 80 to 20% for all roles.

Role Communication. Role communication was measured by examining the electronic forum. 714 messages were sent during the semester, out of which 698 were sent by the ten students (the others were sent by the academic staff). Since there is a load balance with respect to the development task, it is reasonable to refer to the number of messages as indicators for the role part. Table 5.1 provides the message distribution among the role–holders, including their percentage of the total number of messages. In this team an earlier version of the role scheme (see Chapter 2–teamwork) has been used.

As can be observed, the leading group was the most communicative (49.2%) while the customer group was the least (6.5%). Looking into the messages' details, it could have been identified that, indeed, the coach sent a response message for almost every message that another teammate had sent, and both the coach and the tracker also played the role of the continuous integrator, since they felt the need to do so for the project's sake.

There were forum messages that were initiated by the role-holder and others that were responses to other role-holders. Table 5.1 also shows for each role the division of original messages and responses to others. Among students who sent a small number of messages, most of their messages were original and not responses. The students who responded with many messages were actually the core group of students whose roles were the most performed during the semester.

Table 5.1 Forum messages according to roles. (Reprinted from Journal of System Architecture, 52, Dubinsky Y, Hazzan O, Using a role scheme to derive software project quality, 693–699, Copyright (2006), with permission from Elsevier.)

Role	Forum messages (Total)	Forum messages (%)	Forum messages Original + Response
Leading group (49.2%)			
Coach	225	32.2	58 + 167
Tracker	118	17	72 + 46
Customer group (6.5%)			
Customer + End user	32	4.5	19 + 13
Acceptance tester	14	2	13 + 1
Maintenance group (15.2%)			
Presenter + Documenter	53	7.6	26 + 27
Installer	53	7.6	23 + 30
Code group (29.1%)			
Designer	51	7.3	6 + 45
Unit tester	31	4.5	23 + 8
Continuous integrator	33	4.7	25 + 8

Table 5.2 Management level for each iteration(Reprinted from Journal of System Architecture, 52, Dubinsky Y, Hazzan O, Using a role scheme to derive software project quality, 693–699, Copyright (2006), with permission from Elsevier.)

Iteration No.	No. of weeks	No. of messages	Original + responses
1	4	336	119 + 217
2	3	209	125 + 84
3	4	153	85 + 68

Role Management. Role management was calculated for each iteration by summing the total number of messages during all the iteration weeks, and examining their distribution into original messages and responses. Table 5.2 shows the data for the three iterations of the semester.

Figure 5.9 shows the average weekly role management per iteration. Thus, in iterations 2 and 3 the number of original messages was bigger than the responses. This makes sense, since iteration 1 is the launching of the project and of the role scheme, and it requires intensive correspondence. It also seems that iteration 2 is highly managed because of the large number of total messages. This also makes sense, since the last iteration is influenced by the stress of the end of the semester. During the second iteration, the students are more experienced with the project method and perform at their best.

Task

How are the different agile characteristics and elements with which you are familiar at this stage of the book expressed in this case study? Elaborate on each of these characteristics and elements.

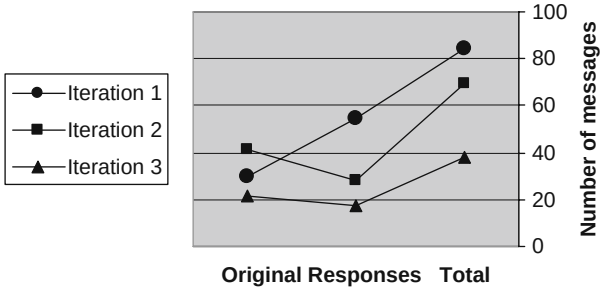


Figure 5.9 *Weekly role management for each iteration. (Reprinted from Journal of System Architecture, 52, Dubinsky Y, Hazzan O, Using a role scheme to derive software project quality, 693–699, Copyright (2006), with permission from Elsevier.)*

5.13 Summary and Reflective Questions

For the specific software project that you currently work on:

1. Set two goals.
2. For each goal indicate three measures that in your opinion can be used to verify that the goal has been met. If the goal is too general, break it down into sub-goals and perform the task for each sub-goal.
3. Suggest work procedures that adhere to the measures you have indicated.
4. Explain how you will use the data from the measurements in order to verify that the goals have been met.
5. Analyze this activity within the HOT—Human, Organizational, and Technological—framework.

5.14 Summary

This chapter deals with measures that suit agile software development environments. Goals are set for the relevant part of the organization, and then measures are set according to the different goals and sub-goals. This way the measures are meaningful and there is a realistic way to add and/or remove measures according to their relevance.

One of the case studies presented in this chapter describes a way to measure the product size by its testability level. This measure demonstrates that "no testing equals no progress." This message is directly connected to the next chapter, about quality.

References

Beck K, Fowler M (2000) Planning extreme programming. Addison-Wesley, Reading, MA

Cohn M (2006) Agile estimating and planning. Robert C. Martin Series, Prentice Hall PTR, Englewood Cliffs

Dubinsky Y, Yaeli A, Feldman Y, Zarpas E, Nechushtai G (2008) Governance of software development: the transition to agile scenario, IT governance and service management frameworks and adaptations. Idea Group Publishing, Information Science Publishing, IRM Press

Hazzan O, Leron U (2006) Why do we resist testing? System Design Frontier 3(7): 20–23

Talby D, Hazzan O, Dubinsky Y, Keren A (2006) Agile software testing in a large-scale project. IEEE Software, Special Issue on Software Testing, pp 30–37

6
Quality

Abstract

High quality assurance is a fundamental element of every engineering process and is considered to be one of the more difficult things to achieve and sustain. Since high quality is also a basic concern of software engineering, there are values and practices that support the assurance of high quality software products and processes. However, these are not sufficient, and in many cases, software products lack the required quality. In this chapter, among other issues, we analyze why this happens. We also describe how quality is perceived using the agile approach, starting with values and practices that support and control the process quality, such as customer collaboration and the planning of a typical agile software development process. We continue with values and practices that support the product quality, for example, refactoring and the feedback gained by exhaustive testing and test automation. Finally, we focus on the Test Driven Development (TDD) practice, analyze its acceptance by software developers and present a way to measure and control TDD processes.

6.1 Overview

In this chapter we describe how quality is assured in the agile approach by describing how the development framework that it inspires encompasses activities that deal with process and product quality. We highlight some of the differences between agile software development and other approaches, and outline why

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_6, © Springer-Verlag London Limited 2008

quality should be defined and how it is employed in agile software development environments.

We distinguish between process quality and product quality and describe how the agile approach refers to them. With respect to process quality we show how the transparency and tightness characteristics of the agile approach increase process quality. For example, the iterative process performed in short iterations of 2–4 weeks, increases process tightness, which in turn raises process quality by enabling better control and faster response to unexpected problems and changes.

With respect to product quality, we describe several agile practices that constantly support keeping high quality products. Among other practices, pair programming enables regular code inspection, and unit testing keeps a testable build and fast identification of integration problems. Specifically, we elaborate in this chapter on one of the agile practices that strongly relates to software quality—test driven development (TDD) (Beck 2003, Feathers 2004, Newkirk and Vorontsov 2004). Though TDD has proven benefits, it is still one of the more difficult practices for implementation by software teams (George and Williams 2003, Meszaros et al. 2003). We analyze this phenomenon and argue that since additional conditions are needed in order to experience the benefits of TDD, it can only partially solve the problems associated with traditional testing. This idea of supporting a practice by other values and practices has already been mentioned in the literature. For example, TDD is strongly supported by other practices, like pair programming and simple design (Beck 2000); software teams are advised to apply TDD only when all teammates agree on its use, and in other cases, it is recommended to give it up (Ambler 2006). Indeed, TDD requires a collaborative development environment and additional supporting practices in order to be integrated successfully into software development processes.

We refer to TDD as a process that, like other processes, should be monitored and controlled. For this purpose, we present a technique called measured TDD, that is based on size and complexity measures and that continuously monitors the TDD process. In addition, we illustrate both how this technique ensures the performance of TDD and how it provides ongoing quality measures.

6.2 Objectives

- Readers will become acquainted with the agile principles and practices related to software quality assurance from the process and product perspectives.
- Readers will learn the practice of test driven development (TDD) and the main issues related to its practice.

- Readers will learn a technique to control the TDD process.
- Readers will learn how to produce quality measures related to the TDD process.

6.3 Study Questions

1. What is quality?
2. What is software quality?
3. Based on your experience, how is software quality expressed in software development environments?
4. What is the difference between process quality and product quality? What is common to these two terms? What connections exist between them?
5. Based on your experience, how would you assure process and product quality?
6. Are you familiar with test driven development (TDD)? If so, how do you experience it, and what are your impressions? If not, describe the way you have performed unit testing till now.
7. How would you suggest maintaining software quality for a specific software project?

6.4 The Agile Approach to Quality Assurance

Some software development methods mimic traditional industries by employing some kind of pipelined production chain. However, the failure of software projects teaches us that such models do not always work well with software development. In order to cope with problems that result from such approaches, the notion of a production chain is eliminated in agile software development environments and is replaced by a more network-oriented development process.

In practice, this means that in agile teams, the task in hand is not divided and allocated to several different teams according to their functionality (for example, designers, developers, and testers), each of which executes its part of the task; rather, all software development activities are intertwined and there is no passing on of responsibility to the next stage in the production chain. Thus, all team members are equally responsible for the software quality. This different concept of the development process results from, among other things, the fact that software is an intangible product, and therefore it requires a different development process

in general, and a different approach towards the concept of software quality in particular, than do tangible products (Hazzan and Dubinsky 2007).

The terminology used by several software development approaches includes two quality-related concepts: Quality and Quality Assurance. The first concept—quality— refers both to the product and to the process. The second concept—quality assurance (QA)—is associated with a specific stage of some software development process and is usually carried out by QA people, who are not the developers of the code whose quality is being examined.

To illustrate the agile software development approach towards quality, we quote Cockburn (2001), who describes quality as a team characteristic: "Quality may refer to the activities or the work products. . . . All checked-in code must pass unit tests at 100 percent at all times. . . . In some cases, quality is given a numerical value; in other cases, a fuzzy value" (118).

Within the framework of agile software development, quality refers to the *entire* team during the *entire* process of software development, and it measures the code as well as the actual activities performed during the development process, both in quantitative and in qualitative manners. Accordingly, the term quality assurance does not appear in agile software development as a specific stage of the development process, and its aims are achieved differently. Table 6.1 summarizes some of the noticeable differences between the attitude towards quality of agile software development methods and of some other approaches (Hazzan and Dubinsky 2007).

Tasks

1. Select three agile practices with which you are already familiar and explain how each of them affects quality. Does it affect the process quality? Does it affect the product quality?

Table 6.1 Some differences between agile and other methods with respect to quality

Quality-related aspect	The agile approach	Some other approaches
Who is responsible for software quality?	All the development team members	The QA team
When are quality-related topics addressed?	All of the time; quality is one of the primary concerns of the development process	At the QA/testing stage
Quality-related activities status	Same as other activities	Low (Cohen et al. 2004)
Work style	Collaboration with all parties	Developers and QA people might have conflicting interests

2. For each of the practices you discussed in Task 1, suggest how it can contribute to quality measurement.
3. Chose one aspect presented in Table 6.1. Based on your experience, elaborate how it is expressed in agile software development environments and in other software development environments.
4. Add aspects to the comparison presented in Table 6.1.

6.4.1 Process Quality

Two main characteristics of agile processes are transparency and tightness (see Chapters 2, Teamwork, and 4, Time). These characteristics require a high quality process, as is illustrated by the following two examples, which are connected to the transparency of agile development processes.

- Planning sessions are performed when all people involved in the development process are present. Specifically, developers and testers, system analysts and the customer, all participate together in the release planning and the planning of each short iteration. This practice causes the project subject and features to be known by all the people involved. Thus, the project's ongoing details are highly transparent. The impact of this high transparency on quality is multifaceted. Among other things, we emphasize the high level communication which significantly decreases misunderstandings and reveals problems before or as soon as they occur. Another impact on quality is the adhering to customer requirements, which are heard by all teammates. This kind of transparency is highly appreciated by teammates and influences positively their morale and the general atmosphere which, in turn, enhance a high quality process.
- Process measures are available all the time to all the people involved in the development process, including the customer. This increases the transparency of all measures, e.g., the project's progress versus estimations, the customer satisfaction, etc. The impact on the process quality of the measurement process itself, as well as its ongoing availability to all the project stakeholders, is clear.

With respect to the process , the following are two examples of agile practices that contribute to high quality process.

- The one day—the Business Day—allocated every two weeks for the presentation of the work accomplished in the previous iteration, for reflective thinking, and for planning the next nine development days, lays out a tight rhythm. This tightness encourages a high quality process, since it controls

the project management, among other ways, by revealing and dealing with unexpected events at early stages.

- The estimated work hours versus the actual time invested in the development task is one measure by which iteration progress, continuous integration, and release progress are measured (see Chapter 5, Measures). This measure is tightly maintained by the reports from each team member with respect to the time invested in the development of each task (estimations are known from the planning session). By letting everyone know the iteration progress and the integration status on a daily basis, and the release progress on an iteration basis, this measure increases the teammates' awareness, care, and attention to process quality issues.

Tasks

1. Suggest additional situations in which transparency and tightness may increase process quality.
2. Suggest situations in which transparency and tightness may decrease process quality.
3. Discuss the characteristics of transparency and tightness with respect to *product* quality.

6.4.2 Product Quality

There is no one standard way to measure software product quality. Besides different measurement approaches, several agile practices, such as the ones presented in what follows, aim at constantly improving the software quality.

- The goal of refactoring is to provide a simple and clear design which is easy to maintain and simplifies future development extensions (see Chapter 8, Abstraction). By performing ongoing refactoring activities, code is kept readable and flexible. The refactoring activities include the examination of the code and its design to find places for improvement, as well as the actual code improvement and change. Such changes also include the necessary modifications in the unit tests that support the code. These changes may be reflected in the code measures that accompany the development. Examples of such measures are the code coverage measure that shows the level of unit tests, or the performance measures that show the level to which the product adheres to the performance requirements. When major needs for refactoring activities are recognized, the refactoring tasks are formulated and entered for consideration in the next planning session, to be presented to the customer and prioritized.

- *Pair programming* enables code inspection for each piece of code (see Chapter 1, Introduction to Agile Software Development). This code inspection provides information in two levels of abstraction. Such a multilevel examination increases the code quality since it enables additional aspects of the code to be captured. Pair programming also ensures adhering to coding standards as well as to unit testing.
- The product quality is also increased by the definition of *acceptance tests* that ensure the validation of each customer story. These acceptance tests are defined by the teammates together with the customer. During the definition process of the acceptance tests, the customer stories are elaborated and become clearer and better understood. The development process of the acceptance test itself increases the teammates' confidence with respect to the correctness of the developed code and allows a smooth presentation of the software functionality at the end of the iteration.

- *Test driven development* (elaborated in the next section) is a technique that enables a step-by-step development of a specific functionality together with its unit test, when each step of the test precedes the respective step of code.

Tasks

1. Summarize the agile perspective on three quality issues that you find interesting. Explain why you selected these issues and share your experience (if any) with respect to your summary.
2. Provide at least two additional agile practices that aim at increasing quality (either process or product quality).

6.5 Test-Driven Development

Test-driven development (TDD) is a programming technique that aims to provide clean, fault-free code (Beck 2003). TDD means that first, we write a test case that fails, and then we write the simplest possible code that enables the test to pass successfully. TDD implies that new code is added only if an automated test has failed. In addition, in order to improve our code, we perform refactoring activities (Fowler 1999), in order, among other reasons, to eliminate duplications. Accordingly, the TDD guideline is red/green/refactor, where red means writing a simple test that fails; green means writing the minimal and simplest code that causes the test to pass (in graphical testing environments this is represented by a

red/green bar displayed when the test fails/passes); refactor means that code quality is improved without adding functionality. This guideline is iteratively implemented in small steps. The cumulative experience of the community is that TDD provides high-quality code (George and Williams 2003), which usually means that the code is readable and includes fewer bugs. Furthermore, there is evidence that through a TDD process, software developers improve their understanding with respect to the developed product (George and Williams 2004).

6.5.1 How Does TDD Help Overcome Some of the Problems Inherent in Testing?

TDD can help overcome some of the common problems associated with traditional testing that are encountered in software projects. Based on Dubinsky and Hazzan (2007), the following TDD analysis addresses cognitive, social, affective, and managerial elements and is structured around arguments frequently offered to explain why, in many cases, traditional testing is skipped. Such arguments are accompanied by explanations on how TDD might help overcome these obstacles.

- *Not enough time to test*—Traditionally, unit testing, if it exists, is performed after the code is written and usually under time pressure. Thus, according to Van Vliet, "the testing activity often does not get the attention it deserves. By the time the software has been written, we are often pressed for time, which does not encourage thorough testing" (Van Vliet 2000, 397). However, "postponing test activities for too long is one of the most severe mistakes often made in software development projects. This postponement makes testing a rather costly affair" (ibid.). Since TDD introduces unit tests throughout the entire development process, this problem is eliminated in TDD processes.
- *Testing provides negative feedback*—Traditional testing processes require the developer to find bugs in his or her own work; in other words, testing activities end in failure. Indeed, who would enjoy that? (Hamlet and Maybee 2001, Hazzan and Leron 2006). In TDD, the rules of the game are reversed. TDD ends in success: after the test fails, code is written and the test passes—success! To illustrate this perspective, we quote the reflection of a developer, Michael Feathers (http://c2.com/cgi/wiki?CodeUnitTestFirst): "Why don't people like testing? Well, the traditional way of testing is tough to take. You write what seems to be perfectly sensible code, then you write a test and the test tells you that you failed. No one wants to hear that. Let's turn it around. Write the test first; run it. Of course it fails .... You haven't

written the code under test yet. Start writing code ... keep testing. Soon, the test will tell you that you've succeeded!"

- *Responsibility for testing is transferred*—There are software development environments in which bugs are found and, in many cases, also fixed by other practitioners than the developer who actually wrote the code; thus, it is not clear who is responsible for each specific testing activity. In TDD processes, the responsibility for testing is borne by the person who writes the code.
- *Testing is a low-status job*—In some software development processes, testing is carried out at the end of the production line, and, as with traditionally working class jobs, the task is assigned low status, which in turn leads to tension among different groups of employees. Cohen et al. (2004) reported that "though most organizations recognize the need for high-quality testers and their specialized skill set, testers still struggle to win the respect they deserve .... The lack of status and support makes the tester's job more difficult and time consuming, as the struggle for recognition becomes part of the job itself" (80). Since in TDD processes all developers test their own code, this negative feeling towards testing is eliminated.

- *Testing is hard to manage*—From a managerial perspective, it is sometimes claimed that in general, testing is a hard process to manage, and that in particular, testing slows down the development process. Since TDD is firmly integrated throughout the entire software development process, it turns development and testing into controlled processes. Furthermore, the fact that TDD is done by writing automatic (not manual) tests, further increases the control level. Indeed, introducing TDD might slow down the development process in the short term simply because testing is actually performed. In the long run, however, it assists in shortening the integration period (especially when performing continuous integration).
- *Testing is hard*—Testing is also difficult from a cognitive perspective mainly because it is not always clear what tests are suitable for a specific purpose and how much testing should be done. The following reflection of a practitioner, Ron Jeffries, explains how TDD supports the testing from the cognitive perspective (http://c2.com/cgi/wiki?RonJeffries): "A key aspect of this process: don't try to implement two things at a time, don't try to fix two things at a time. Just do one. When you get this right, development turns into a very pleasant cycle of testing, seeing a simple thing to fix, fixing it, testing, getting positive feedback all the way. Guaranteed flow." Being a detail-oriented and explicit process, TDD improves one's understanding of what should be developed, since the test must be written prior to the writing of the code.

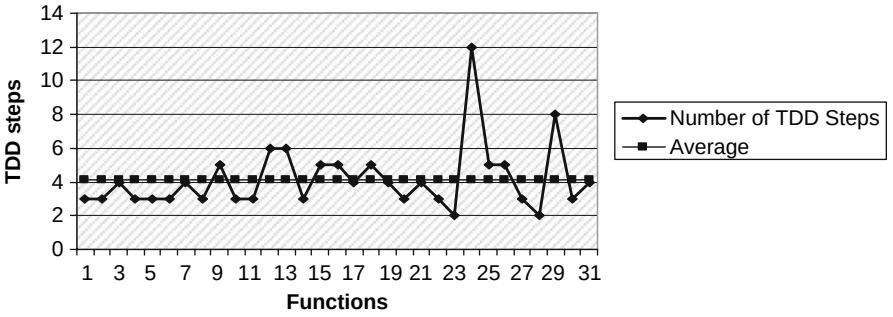
Task

Represent the above TDD analysis within the HOT—Human, Organizational, and Technological—framework.

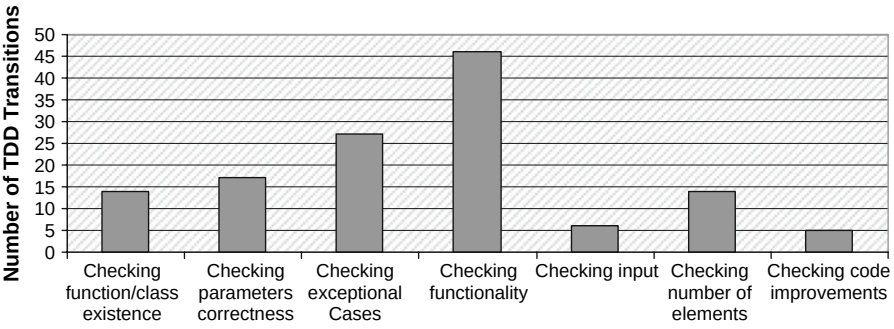
Though TDD helps cope with traditional problems related to traditional testing, it is, however, not sufficient and is still not performed at full scale. The following are two illustrative case studies that show that developers still find it hard to implement the entire TDD process, including the refactoring stage. These case studies, as well as other data, motivated the development of the measured TDD technique, presented in the following part of this chapter.

6.5.2 Case Study 6.1. TDD Steps

This data set looks at the activities performed during the development of specific functions developed by TDD as part of a large software project (Dubinsky and Hazzan 2007).



(a)



(b)

Figure 6.1 *TDD steps and the reasons for the respective TDD transitions.*

An examination of 31 functions, developed using TDD, reflects a total of 129 TDD steps. The participants were asked to save the file that corresponds to each TDD step, to explain why they moved on to the next step, and to document their refactoring activities. By refactoring activities we refer to code activities that should be carried out frequently, on the level of the currently-developed unit, each time after several test and code steps have been developed.

Figure 6.1a presents the number of TDD steps for each of the 31 functions, as well as the average number of TDD steps per function (horizontal, dashed line), which was found to be 4.16. Figure 6.1b presents the reasons given for making TDD transitions to the next step in the development of the examined functions, as reported by the participants, as well as the number of times each reason was given.

As can be observed, the two main reasons for moving to the next TDD step are checking the functionality of the feature to be implemented and checking exceptional cases. An examination of the participants' reports on their refactoring activities revealed that the participants did not perform code refactoring at all, i.e., they did not stop to improve the unit code each time after several test and code steps had been developed; rather, they continued developing till the unit coding was completed. We note that refactoring activities that are on the project level, such as improving class hierarchies, are not included in the analysis presented here.

6.5.3 Case Study 6.2. Reflection on TDD

Participants were asked to reflect on their *first* TDD experience. Following are some of the participants' expressions, categorized into pros and cons.

Developers described the advantages of as follows:

- "It makes us think ahead."
- "There are less bugs. Developers are forced to produce high-quality software."
- "It helps us get acquainted with the software components."
- "It makes us think before coding."
- "It requires writing minimum code in order to let the tests pass."
- "It saves time that used to be dedicated to bug finding."
- "It helps in quality assurance!"

Developers described the disadvantages of TDD as follows:

- "Work is delayed because of relatively simple items."
- "It doubles the time to write code."

- "It increases development time."
- "There is no global view when dealing with complicated components."
- "It is hard to identify the critical cases."
- "It is not suitable for every kind of task."
- "It is a waste of time if the code is not used later."

An examination of these reflections reveals two main observations. First, developers tend to refer to TDD as a thinking activity in general, and as a thinking-before-coding activity in particular. This observation means that when TDD guides the development process, coding is not perceived by developers as a spontaneous developer-computer interaction, but rather as an activity that requires thinking before performing. This can be explained by the fact that unlike TDD, which forces the developer to think before coding, there are cases in which developers tend to start coding intuitively.

The second observation involves the contradictions and conflicts that TDD introduces. For example, one developer claimed that TDD ensures that fewer bugs occur and consequently leads to shorter integration times. At the same time, however, this developer claimed that the time overhead that TDD introduces is a disadvantage. Another developer claimed that since she thinks before coding, she knows exactly what she is going to code. At the same time, however, this developer indicated a feeling of uncertainty when practicing the TDD approach. A third developer claimed that TDD disrupts the coding continuity, but acknowledged its convenience. These contradicting reflections may be explained by the fact that, traditionally, developers used to code first and test later; TDD forces them to perform these activities in a reversed order. Accordingly, their first TDD experiences cause mixed feelings and contradicting opinions.

Tasks

1. Explain the source of the phenomena presented in Case Studies 6.1 and 6.2.
2. Have you experienced similar situations and/or feelings to the ones presented in these case studies?

Based on the above illustration, it is clear that TDD has many benefits and that it might indeed help cope with some of the cognitive, affective, social, and managerial problems associated with traditional testing. However, Case Study 6.1 and the accumulated experience of the agile community tell us that though

TDD does help overcome many of the problems associated with traditional testing processes by providing a tight and clear testing procedure to follow, it is not fully performed in agile projects and is still considered to be one of the more difficult practices to introduce when the decision to apply the agile approach in the organization is taken. Furthermore, even when TDD is applied, developers tend to reduce the number of TDD steps and to skip the refactoring phase, which is required repeatedly after every few TDD steps. In addition, according to Case Study 6.2, the new work habit that TDD introduces leads to some confusing feelings.

One possible reason for these phenomena is that, like other processes that must be measured, disciplined, and controlled, TDD processes should also be measured and controlled (see Chapter 5, Measures); that is, their tightness should be increased (see Chapter 4, Time). Accordingly, measures should be taken alongside the TDD steps to lead and guide this process. To that end, a technique— measured TDD—whereby two measures are added to the TDD process, is presented, rendering it a more controlled process.

6.6 Measured TDD

The measured TDD technique (Dubinsky and Hazzan 2007) aims at improving the performance of TDD by incorporating measures and control elements into the TDD process itself. Specifically, at the end of each TDD step, developers measure the size and complexity of the developed code. Size is determined by the number of lines, and complexity is determined by calculating the cyclomatic complexity (McCabe 1976, Watson and McCabe 1996), whereby a sequential method has a complexity of 1, and each decision that causes a binary split adds 1 to the complexity[1]. Size and complexity are measured also with respect to the evolved *test*. Other measures can be taken as well. However, we focus on the aforementioned measures since they are simple, easy to use, and can be taken automatically.

Measured TDD has the added value of measuring while developing. Specifically, the use of the size and complexity measures of both the test and the code helps developers determine when, while implementing TDD steps, they should refactor the code. This phenomenon is reflected in Case Studies 6.3 and 6.4.

[1] The Metrics software, for example, provides a plug-in that automatically calculates McCabe cyclomatic complexity (http://metrics.sourceforge.net/).

6.7 Quality in Learning Environments

In this studio meeting the students complete the development tasks and integrate them. Naturally, this task is easier for teams who apply continuous integration and high level testing than for teams which do not. They also prepare the presentation of the first iteration to be presented to the customer in the next studio meeting.

We focus in this section on case studies taken from the academic setting to show how students work with measured TDD.

6.7.1 Case Study 6.3. Size and Complexity Measures

An examination of the size and complexity measures of 19 different functions developed through a measured TDD process and of the 75 TDD steps associated with these 19 developed functions reveals the following observations (Dubinsky and Hazzan 2007).

First, since in general each line of code is inspected by several tests (and each test is usually one line long), more lines of test are expected than lines of code. Indeed, the 19 different functions developed through the measured TDD process yielded about two times more lines of tests (1582) than lines of code (800).

Second, as can be observed in Figure 6.2, which presents the size (6.2a) and complexity (6.2b) of the code for each measured TDD step, most of the functions were developed in three or four measured TDD steps. Figure 6.2a shows that most functions have less than 20 lines of code, which means that the functions remain short and simple. Figure 6.2b, which presents the code complexity of each measured step in terms of cyclomatic complexity, further validates this assumption about the nature of the functions developed through the measured TDD process.

Third, though the number of TDD steps did not increase relative to Case Study 6.1 (the average number of measured TDD steps for these 19 functions was 3.95), we will observe later the added value of the measured TDD mainly by the actual performance of refactoring activities that lead to simpler code. We note that when the data presented in Case Study 6.3 are combined with another data set that refers to 16 functions developed through a measured TDD for which the average number of steps was 5.5 (an increase of 32% relative to Case Study 6.1), the average number of steps for the 35 functions is 4.66, which indicates a 12% increase relative to Case Study 6.1.

Task

Why, in your opinion, does measured TDD increase developers' awareness of refactoring activities?

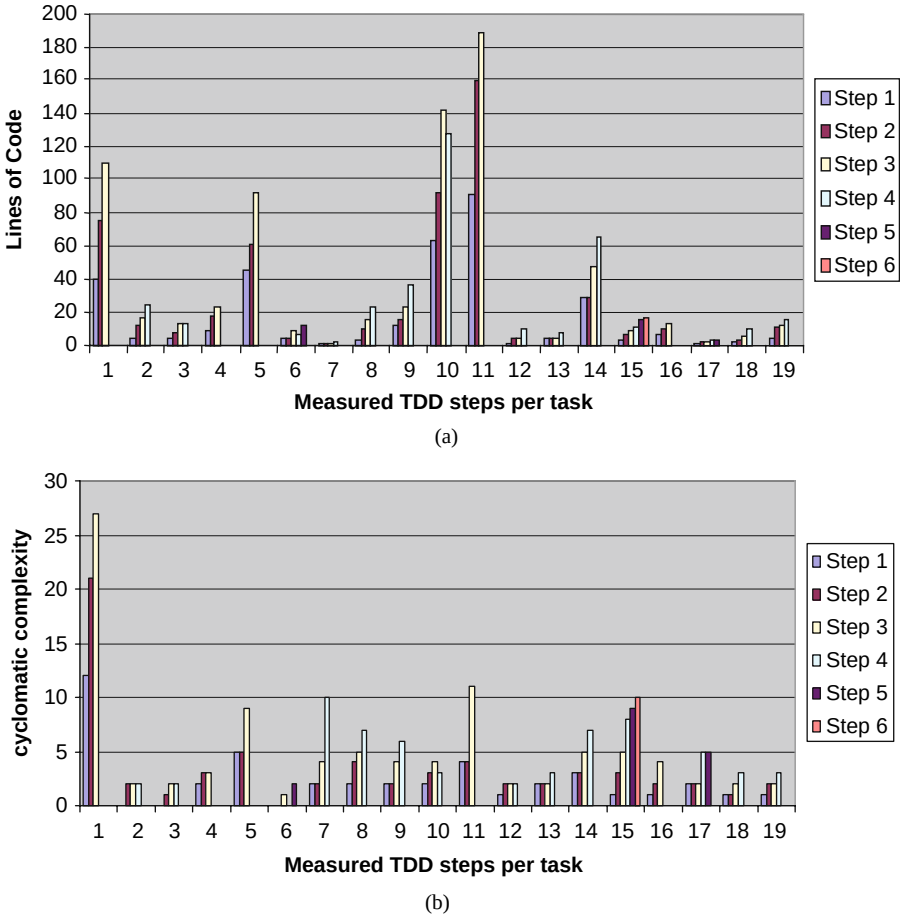


Figure 6.2 *Code size and complexity for measured TDD steps.*

Fourth, Figure 6.2b reveals that the cyclomatic complexity in most cases is less than 5. As mentioned before, this means that most of these functions are not complicated. In one case, for example, in which the cyclomatic complexity soared 27, the code was checked and it was found that the task included nine hash table manipulation functions. When complexity was higher than 5, developers suggested improvements and in some cases also implemented them. In two of the 19 cases in which the cyclomatic complexity was reduced at some stage (#6 and #10), the size of the code was also reduced (see Figure 6.3). This indicates that since developers constantly monitor their work, measured TDD keeps complexity, as well as the size of the code, low.

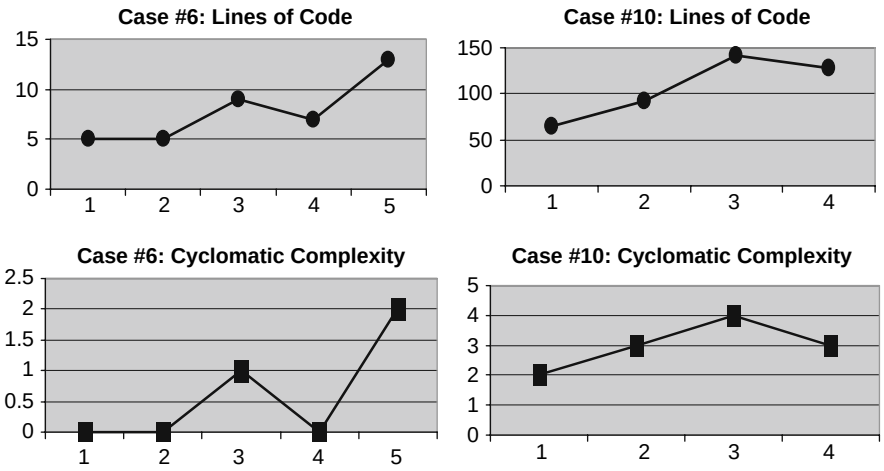


Figure 6.3 *Reduction in cyclomatic complexity and code size.*

6.7.2 Case Study 6.4. Illustrating Measured TDD

This part illustrates the measured TDD process by a Java class that was developed using the measured TDD technique.

The project deals with the development of a shell language that enables search in several digital libraries (Dubinsky et al. 2006). In order to increase awareness of measures as well as of their implications, the developers were asked to complete a containing the following information: step number, test and code descriptions, size of test and code, cyclomatic complexity of test and code, and a description of the refactoring activities performed. The developers were told that they must complete all table columns for each TDD step and, specifically, a refactoring description must be written for all steps, even if they decide that a specific step requires no refactoring.

The name of the class was *SearchCommand*. Table 6.2 (Dubinsky and Hazzan 2007) presents a tracking table for the class development, as constructed by one of the developers. As can be observed, the table was indeed used by the developer to track the development process. The refactoring column was completed for all 12 steps and indicates the exact step (#7) where the developer became aware of the need to refactor. At this stage, the developer also started to make use of the measures, citing the high as the rationale for her need to refactor. Though the need to refactor was detected in Step #7, the developer decided to continue with the class development before performing the refactoring (see the Comments

column). The refactoring of both code and test was carried out in the last two steps (#11 and #12).

Figure 6.4 presents the test and code measures as reported in the tracking table (Table 6.2). (Dubinsky and Hazzan 2007) It is clear that the refactoring activities, of both the code and of the test, reduced the size and lowered the size and lowered the complexity of both code and test. It can be concluded that, as a result, the clarity and simplicity of both code and test increased.

To illustrate how measured TDD is used, we present the code for three of the steps (#1, #7, and #12). The test of Step #1 consists of a sanity check. The developer checks that *SearchCommand* can be instantiated. Naturally, this test fails since no code exists at that time (note that the developed class extends the abstract class *ACommand*, and therefore three empty methods are created). Both measures are low and no refactoring action is needed.

The examination of Step #7 shows that the code indeed became longer and now includes many repetitions. The code complexity measure for this step is 13 and increases to 19 (step #10) before refactoring is performed (see Table 6.2).

Finally, in Steps #11 and #12, is performed. Both test and code are improved by introducing a method that eliminates code duplications. The measures indicate that the code is indeed more concise and simpler (see Table 6.2 and Figure 6.4). These three stages of the refactored code are presented in what follows.

Code of Step #1:

```
package gsdl.command;
import gsdl.exception.*;
public class SearchCommand extends ACommand {
public void setParameter(String name, String value) throws Illegal-
CommandException {
     }
     public void verifyParameters() throws IllegalCommandException {
     }
     public void execute() throws ScriptException {
     }
}
```

Code of Step #7 (Since this is a printout, a difference exists between its number of lines and the student's count of the lines of code as appeared in Table 6.2):

```
package gsdl.command;
import gsdl.exception.*;
```

Table 6.2 The measured TDD tracking table for the class *SearchCommand*

#	Test	Code	Test lines#	Code lines#	Test CC	Code CC	Refactor	Comments
1	Sanity check	Method signature	11	3	1	1	Not needed—I didn't start yet	Method signature and test's tearUp method
2	Set parameter—text	Handle text parameter	19	9	4	3	Not needed	
3	Set parameter—name	Handle name parameter	31	14	6	5	Not needed	
4	Set parameter —search List	Handle search List parameter	40	19	8	7	Not needed	
5	Set parameter —meta	Handle meta parameter	51	26	10	9	Not needed	
6	Set parameter—op	Handle op parameter	62	31	12	11	Not needed	
7	Set parameter—case Sensitive	Handle case Sensitive parameter	74	36	14	13	CC high -> need refactoring, later	Needed refactoring, I'll do it at end
8	Set parameter—rank	Handle rank parameter	84	41	16	15	CC high -> need refactoring, later	Needed refactoring, I'll do it at end
9	Set parameter —intoResultList	Handle intoResult List parameter	95	46	18	17	CC high -> need refactoring, later	Needed refactoring, I'll do it at end
10	Set parameter—popup	Handle popup parameter	106	51	20	19	CC high, code HIGH -> need refactoring	Finished method, now refactoring

Table 6.2 (continued)

#	Test	Code	Test lines#	Code lines#	Test CC	Code CC	Refactor	Comments
11	Refactor existing code	Refactoring by introducing checkValue function, it will reduce both CC and CodeLines	106	26	20	10	Introduced a method that reduced code duplication	
12	Refactor test code	Refactor test code by introducing a help test method	16	26	2	10	Introduced a method that reduced duplication in test code	

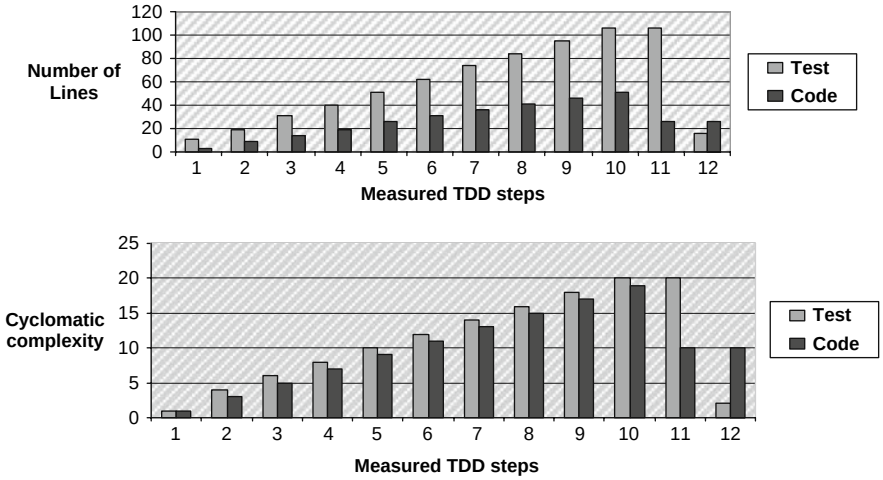


Figure 6.4 *Test and code measures of SearchCommand.*

```
public class SearchCommand extends ACommand {
private static final String PARAM_TEXT = ‘‘-text’’;
private ... [variables declaration and initialization]
public void setParameter(String name, String value) throws Illegal-
CommandException {
     if (PARAM_TEXT.equals(name)) {
            if (value == null || value.length() == 0)
             throw new IllegalCommandExcep-
tion(STR_GOT_NULL_VALUE + name);
            } else if (PARAM_NAME.equals(name)) {
            if (value == null || value.length() == 0)
            throw new IllegalCommandExcep-
tion(STR_GOT_NULL_VALUE + name);
            else
            strName = value;
     } else if (PARAM_LIST.equals(name)) {
            if (value == null || value.length() == 0)
            throw new IllegalCommandExcep-
tion(STR_GOT_NULL_VALUE + name);
            else
                  strSearchList = value;
```

```
            } else if ... [Same for more PARAM_'s]
                    ..
            }
            else throw new IllegalCommandException("Got invalid
parameter: `` + name);
}
public void verifyParameters() throws IllegalCommandException { }
public void execute() throws ScriptException { }
}
```

Code of Step #12:

```
package gsdl.command;
import gsdl.exception.*;
public class SearchCommand extends ACommand {
private static final String PARAM_TEXT = ``-text'';
private ... [variables declaration and initialization]
public void setParameter(String name, String value) throws Illegal-
CommandException {
    checkValue(name, value);
    if (PARAM_TEXT.equals(name)) {
    } else if (PARAM_NAME.equals(name)) {
             strName = value;
    }else if (PARAM_LIST.equals(name)) {
             strSearchList = value;
    }else if ... [Same for more PARAM_'s]
            ...
    }
    else throw new IllegalCommandException(``Got invalid
parameter: '' + name);
}
public void verifyParameters() throws IllegalCommandException { }
public void execute() throws ScriptException { }
private void checkValue(String name, String value) throws Illegal-
CommandException {
     if (value == null || value.length() == 0)
             throw new IllegalCommandExcep-
tion(STR_GOT_NULL_VALUE + name);
}
}
```

6.7.3 Teaching and Learning Principles—The Case of Quality

Naturally, software engineers should be educated for quality. This message is clearly delivered by the Software Engineering 2004 volume of the Computing Curricula 2001 (http://sites.computer.org/ccse/), in which Software Quality is one of the Software Engineering Education Knowledge Areas (p. 20), and is described as follows: "Software quality is a pervasive concept that affects, and is affected by all aspects of software development, support, revision, and maintenance. It encompasses the quality of work products developed and/or modified (both intermediate and deliverable work products) and the quality of the work processes used to develop and/or modify the work products. Quality work product attributes include functionality, usability, reliability, safety, security, maintainability, portability, efficiency, performance, and availability" (31).

Furthermore, the Software Engineering Code of Ethics and Professional Practice, formulated by an IEEE-CS/ACM Joint Task Force, addresses quality issues and outlines how software developers should adhere to ethical behavior. In Chapter 9, Trust, we discuss the subject of ethics and the principles of the above code. At this stage, we highlight Principle 3 of the code, which focuses on quality: "3 PRODUCT—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible."

Based on the assumption that the concept of quality should be taught as part of software engineering education, the question that should be asked is how quality should be taught.

In general, the nature of the software development methods that inspire a curriculum, is usually reflected in the curriculum itself. Accordingly, when the concept of quality is integrated into a software engineering program that is inspired by agile software development, quality-related issues should be integrated and intertwined in all learned topics. This idea is illustrated in what follows, in which the teaching of quality issues, as they are perceived by the agile approach, is compatible with two of the teaching and learning principles, presented in Chapter 14, Delivery and Cyclicality:

- *Let the learners experience the software development approach*—(see Teaching and Learning Principle 2 in Chapter 1)—Since quality is a complex concept, its gradual learning process should be based on the learners' experience. With respect to acceptance testing, active learning can be expressed in several ways. First, learners are active in the definition of the software requirements. Second, learners define the acceptance tests together with the customer and verify that they meet the requirements. Third, they develop the acceptance tests and use them to validate the developed code. And fourth, they are encouraged to reflect both on each individual step and on the entire process.

- *Elicit* communication—(see Teaching and Learning Principle 5 in Chapter 3)—This principle can be applied very naturally in the context of quality since it is a multifaceted concept. During the learning of quality activities, learners can be asked to identify its different facets, to allocate the learning of its parts to different team members—first learning them, and then subsequently teaching them to the other team members in the stage that follows. In the spirit of agile software development, it is appropriate to assign the different parts that are to be learned to pairs of learners (rather than to individuals), in order to foster learning processes. When the team members present what they have learned to their teammates, not only do they share their knowledge, but further communication is enhanced.

6.8 Summary and Reflective Questions

1. Select two of your development tasks and develop them using the measured TDD technique. Construct a tracking table.
2. Based on this experience, evaluate the final code of these two tasks with respect to readability and meeting the required functionality. In addition, evaluate the final tests with respect to the coverage they supply to the developed code.
3. Would you recommend your colleague to use the TDD technique? The measured TDD technique? Why?
4. Suggest how unit tests and acceptance tests can assist in the evaluation of software projects in general and their code in particular.
5. Present the different ideas presented in this chapter within the HOT—Human, Organizational, and Technological—framework.

6.9 Summary

This chapter describes the agile approach to process and product quality. Specifically, it delves into the details of TDD implementation. We present a technique for function/unit development that uses size and complexity measures for monitoring and controlling the TDD process. This technique helps overcome difficulties developers face in applying and sustaining TDD, and further encourages

function refactoring; thus the TDD advantage of developing high quality software is enhanced. The simple and easy-to-automate measures ensure no significant overhead. This technique also provides a means to promote automated unit tests, which are considered to be one of the basis steps towards the production of high quality products.

References

Ambler SW (2006) Introduction to test driven development (TDD). http://www.agiledata.org/essays/tdd.html. Last updated July 28, 2006

Beck K (2000) Extreme programming explained. Addison-Wesley Reading, MA

Beck K (2003) Test-driven development by example. Addison Wesley, Reading, MA

Cockburn A (2001) Agile software development. Addison-Wesley, Reading, MA

Cohen CF, Birkin SJ, Garfield MJ, Webb HW (2004) Managing conflict in software testing. Commun ACM 47(1):76–81

Dubinsky Y, Catarci T, Kimani S (2006) Active data and the digital library shell. Joint conference on digital libraries (JCDL) workshop on digital libraries in the context of users' broader activities. Chapel Hill, NC, USA

Dubinsky Y, Hazzan O (2007) Measured test-driven development: using measures to monitor and control the unit development. J Comput Sci 3(5):335–344

Feathers M (2004) Working effectively with legacy code. Prentice Hall, Englewood Cliffs

Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley Professional, Reading

George B, Williams L (2003) An initial investigation of test driven development in industry. Proceedings of the ACM symposium on applied computing, March 9–12, Melbourne, Florida

George B, Williams L (2004) A structured experiment of test-driven development. Inform Software Tech 46:337–342

Hamlet D, Maybee J (2001) The engineering of software. Addison Wesley, Reading, MA

Hazzan O, Leron U (2006) Why do we resist testing? System Design Frontier 3(8):13–17.

Hazzan O, Dubinsky Y (2007) Teaching agile software development quality assurance. In: Stamelos I, Sfetsos P (eds) The agile software development quality assurance book. Idea Group Inc., Chap. IX:171–185

McCabe T (1976) A complexity measure. IEEE T Software Eng 308–320

Meszaros G, Smith SM, Andrea J (2003) The test automation manifesto. Proceedings of the XP/agile conference, pp 73–81

Newkirk JW, Vorontsov AA (2004) Test-driven development in Microsoft .NET. Microsoft Press

Van Vliet H (2000) Software engineering—principles and practice. Wiley, New York

Watson AH, McCabe TJ (1996) Structured testing: a testing methodology using the cyclomatic complexity metric. NIST Special Publication 500–235

7
Learning

Abstract

Software development is a learning process. This statement can be explained both from the customers' and team members' perspectives. At the beginning of the development process customers do not know explicitly and entirely what their requirements for the developed product are, but improve their understanding with respect to these requirements during the development process. The team members keep improving their understanding of the customer requirements as well as of the developed product. If software development is a learning process, an appropriate learning environment should be provided to all project stakeholders. Indeed, this is another characteristic of agile software development environments—they support leaning processes. This aspect is explored in this chapter by an examination of agile software development environments from the constructivist perspective.

7.1 Overview

HOT

In this chapter we explore how agile software development environments support learning processes of agile teams and customers of software products. This assertion is derived from the fact that the development of a software product can be viewed, among other ways, as a learning process for all the project's stakeholders. The customer learns in a gradual way what specific features will answer his or her needs; team members learn the customer's requirements as well as how those requirements can be implemented in the best way, taking into consideration the

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_7, © Springer-Verlag London Limited 2008

software characteristics, such as quality, readability, simplicity, and code manageability; the organization management learns the business environment with which it deals with respect to a given project and steers the organization accordingly.

The topics that should be learned during a typical software development process are not simple. Most of them are unknown, abstract, complex, and multifaceted. Therefore, learning tools should be provided to software teams. This chapter explores mechanisms that agile software development environments provide for the stakeholders of agile projects to ease and support their learning processes.

The very same idea is applied with respect to learning this book. In the learning section of this chapter we explain this assertion and present a set of reflective activities to promote the book learning process.

7.2 Objectives

- Readers will learn how agile software development supports learning processes.
- Readers will become familiar with the constructivist approach to learning processes and how agile software development can be analyzed from this perspective.
- Readers will experience reflective processes in agile teams at team and individual levels.
- Readers will get a holistic picture of the main ideas presented so far in this book.
- Readers will grasp the relationships between the main ideas presented so far in the book.

7.3 Study Questions

1. In your opinion, what are the three main topics related to agile software development discussed so far in the book? Why did you identify these topics as the main agile topics? Why are they agile?
2. Identify the main connections and interrelations between the different topics reviewed so far in the book. Why did you indicate them as the main connections and interrelations?

3. Analyze the development process of the software you currently develop. Identify the three main characteristics of your development process. What are the three advantages of that process? What are the three main pitfalls of that process? How can the process be improved?
4. Reflect on the development process of your current project. Do you recall situations in which you had to change the process based on new understandings you gained during the process? Describe these cases: What exactly led you to change the process? How did you change the process? What were the consequences of this change in the process?
5. Based on your current understanding of agile software development, define the concept "agile software engineering."
6. What topics, in your opinion, should be presented in the rest of this book to get a fuller picture of agile software development?

7.4 How Does Agile Software Development Support Learning Processes?

7.4.1 Agile Software Development from the Constructivist Perspective

Constructivism has been mentioned in the second teaching and learning principle presented in Chapter 1, Introduction to Agile Software Development.

Constructivism is a learning theory that examines the nature of learning processes. A central tenet of the constructivist approach is that learners construct new knowledge by rearranging and refining their existing knowledge. More specifically, according to the constructivist approach, new knowledge is constructed *gradually,* based on the learner's existing mental structures. Mental structures are developed in steps, each elaborating on preceding ones, though there may of course be regressions and blind alleys. This process is referred to by Leron and Hazzan (1997) as "learning by *successive refinement*" and it is closely related to the Piagetian mechanisms of assimilation and accommodation (Piaget 1977). The term *successive refinement* itself is borrowed from computer science, where it refers to a methodology that guides a gradual elaboration of complex programs (Dijkstra 1972). This methodology is based on the assumption that successive refinement is an especially effective way for the human mind, with its particular strengths and limitations, to deal with complexity.

Since software development is a complex process (Hamlet and Maybee 2001), methods that support learning processes should be provided to all software projects' stakeholders—team members, customers, and management. According to the constructivist perspective, these tools should support a gradual learning process, through which the project stakeholders improve their understanding regarding the development process and product. These means should on the one hand lead to improved understanding, and on the other hand enable mistake corrections in an easy and inexpensive way. The fact that a software development method provides these means indicates that it legitimates learning processes and misconception correction. Indeed, agile software development legitimizes such processes. (Eckstein 2004) says with respect to this that "accepting and embracing change also means understanding that errors are part of the process" (122).

In what follows we explain how agile software development supports the gradual construction of knowledge related to the development of software systems. The discussion is based on the practice of short iterations. It is shown how short iterations lead to improved understanding of the developed product by the customer and team members, and consequently, they are able to carry out the software development task more confidently.

7.4.2 The Role of Short Releases and Iterations in Learning Processes

One of the main practices of agile software development processes is short releases and iterations. An iteration of an agile software development process includes all the phases and activities involved in software development processes; they are applied, however, only on a portion of the developed software product. The part on which an iteration focuses is determined by the customer, who prioritizes the development tasks according to his or her preferences.

This practice of agile software development has been mentioned several times in this book so far, especially in Chapter 3, Customers and Users. Different aspects of this practice have been highlighted in these discussions. In the context of this chapter we highlight the idea that the fact that the software is developed in short iterations and releases guides the customer, as well as the team members, in a gradual process of knowledge construction with respect to the developed software.

It is a known fact that customers face difficulties in determining in advance all of the required features of the software. Still, some software developers request customers to define their software requirements in detail at the beginning of the development process. Agile software development, however, takes a different approach: iteration and release Business Days—which include planning and reflective sessions—are conducted frequently in accordance with the practice of

short iterations and releases. These planning sessions, and the reflective processes that accompany them, provide the customers with the opportunity to rethink, refine, and improve their understanding of the software they require. Consequently, customers are able to define and communicate their requirements to the software team members in a more precise and clear manner.

In addition, in each short iteration and release the team members get feedback with respect to their understating so far of the customer's requirements. If they misunderstand the requirements, the customer can clarify his or her intentions; if they do not understand a specific customer request, they have the opportunity to clarify the customer's intention in a face-to-face interaction.

Indeed, this kind of interaction is based on the realization that misunderstanding exists in dealing with customer requirements and that an opportunity to improve and correct the understanding of what should be developed, both by the customer and the software team members, should be provided.

Tasks

1. Describe a scenario in which a customer improves his or her understanding of the product requirements through three or four iterations. Indicate the specific elements that caused and led the customer's learning process. In your description address also the kind of interaction that influenced this learning process.
2. How does such an approach at learning processes influence the culture and social interactions among the different project stakeholders?

From the constructivist perspective, a development process that is based on short iterations has several benefits which are directly connected to learning processes.

First, it allows both the customer and the software team members to focus on a relatively small part of the iteration.

Second, it allows the customer and the software team members to gradually improve their understanding with respect to the developed software. This is because short iterations do not require dealing with future developments that are unknown at a specific stage, and that will probably be clarified later when the development proceeds.

Third, short iterations improves communication between among the project stakeholders in general and between the customer and the team in particular. The Business Day, which takes place after each short iteration, and in which the customer, the team, and management participate, enables all the project stakeholders to gather together, communicate, become familiar with the others' perspectives on the project, express their concerns with respect to the development

process and the product, and reflect on previous developments. All these activities improve the understanding of the development process and the product by all the project stakeholders.

Fourth, short iterations define very clearly the time for feedback and reflective sessions. Feedback is provided by the customer at the end of each iteration; reflective sessions also take place at the end of each iteration.

Fifth, in addition to the lessons learned during such reflective sessions, such a break enables the developers to rest and detach for a while from the demanding, complex, and tight process of software development. It enables the team members to exploit their capabilities in a better way when they return to the development tasks of the next iteration.

Sixth, short iterations foster courage to raise problems and to attempt to solve them together. At the end of each iteration the team presents to the customer what has been accomplished during the last iteration, and if needed, shares with the customer any misunderstandings and/or problems in the development process.

Tasks

1. Analyze the practice of short releases within the HOT—Human, Organizational, and Technological—framework.
2. What other agile practices that have been reviewed so far in the book support learning processes?

Another agile practice that supports learning processes is reflective thinking. This thinking mode will be reviewed in depth in Chapter 11, Reflection, and Chapter 14, Delivery and Cyclicality. It is, however, important to begin this practice even before its theoretical background is learned. In the next section, which addresses learning in learning environments, a set of reflective tasks on learning this book is presented.

7.5 Learning in Learning Environments

In the first part of this chapter we explain how agile software development environments support learning processes by laying out a development process that encourages gradual learning. This gradual learning perspective is applied also with respect to learning this book and the course for which the book is used. Therefore, in what follows, we use the terms course and book alternately.

Accordingly, and in the spirit of the agile perspective, this chapter ends the book's first iteration and opens the second iteration. It reviews what has been learned so far in the course and what will be taught in upcoming lessons. The rationale for this intermediate step is to summarize the ideas presented so far in the book into one comprehensive picture that captures the spirit of agile software development environments and, at the same time, serves as an introduction for the second and third iterations of the course, which on the one hand delves into more detail, and on the other hand presents agility in a wider context.

This intermediate step is based on reflective processes (see Chapter 11, Reflection). Reflective tasks are offered to all the book's readers and course participants—software developers, students, academic coaches, and instructors—in different settings: groups, development teams, and individuals. Such an iteration-based process enables us to look back at what we have done, reflect on what has been accomplished, and realize what we have achieved. In addition, such a process sets the stage and infrastructure for the next iterations.

At this stage, we stop learning new agile ideas, check our understanding of what we have learned so far in the book, reflect on what we have learned so far, clarify issues that are unclear to us at this point, and outline what will be learned in future chapters.

7.5.1 Gradual Learning Process of Agile Software Engineering

We first review how each of the ideas presented as the rationale for short iterations with respect to software development is reflected in the learning process of this book.

First, learning in iterations allows the learners to focus on a smaller part of the course content. This ability is important in the learning process of complex ideas, such as software engineering. Such a teaching process also encourages the instructor not to worry if not all course topics and aspects are addressed at each stage. This is because it will be possible to revisit these topics and add details in the next iterations of the learning process.

Second, short iterations allow learners to gradually improve their understanding of the learning material, based on the previous lessons and the actual software development in the studio, if it is carried out.

Third, the practice of short iterations improves communication between the instructor and the students. After each short iteration, the instructor stops the teaching of new ideas (as we do now), has an opportunity to hear the students' voices (as we will do later in this chapter), communicates with the students, and becomes familiar with what is interesting for the students, how they understand

the material taught so far, and what bothers them. Based on new understandings the instructor gains, he or she can consider how to improve the continuation of the course. From the students' perspective, this break enables them to share with the instructor what topics they find interesting and relevant, what ideas are difficult for them to grasp, and so on. Clearly, such a dialogue influences the course's atmosphere and inspires a culture that is motivated by a shared goal.

Fourth, if the course is accompanied by the development of a software product in the agile approach, the end of each short iteration serves as an opportunity to present to the academic coach and to the project customer what has been accomplished during the last iteration. It is also an opportunity to share with the instructor and the academic coach problems in the development process (if any exist), and guided by the academic coach, to sketch together the future road map of the development process.

Fifth, short iterations define very clearly the timing for feedback and reflective sessions. Lessons learned based on the iteration elicited in the reflective process, as we will do later in this chapter, can start being applied in the next iteration.

Finally, in a very similar way to software developers in the industry, after a short break dedicated to reflective sessions, feedback, and learning processes, the learners return with new energy to the course in general, and in particular to the development of the next iteration of the software product, if one accompanies the course learning.

7.5.2 Learning and Teaching Principle

We add now Learning and Teaching Principle 4, which deals with the integration of reflective sessions in the course. The full list of Learning and Teaching Principles is presented Chapter 14, Delivery and Cyclicality.

7.5.2.1 Teaching and Learning Principle 4: Elicit Reflection on Experience

The importance of introducing reflective processes into software development processes in general, and into agile software development processes in particular, has already been discussed (Derby et al. 2006, Hazzan 2002, Hazzan and Tomayko 2003, Kerth 2001), based mainly on Schön's *Reflective Practitioner* perspective (Schön 1983, 1987).

According to this principle, learners should be encouraged to reflect on their process of learning the taught software development approach. Reflection

processes should not be limited to technical issues, but rather should also address feelings, work habits, and social interactions related to the software development process.

Based on this principle, after each studio meeting the students are asked to reflect on some aspect of the development process, such as their role in the process, their experience with specific activities, and their teamwork conception.

7.5.3 The Studio *Meeting—End of the First Iteration*

So far, till mid-semester, the students have developed one iteration in the studio and have two more iterations to go in order to complete the first product release. The first iteration lasted seven weeks and included the learning of the project subject and development method and the definition and construction of the project development environment. The second iteration, which starts at this meeting, lasts four weeks—one week for high-level design and three weeks for development work. The product will be presented to the customer in the eleventh week of the semester. The third and final iteration of the product development, will include three weeks—one for high-level design and two weeks of development—and will be presented in the fourteenth and last meeting of the semester. This project timetable clearly reflects agility as it is applied in real-life situations, from both the team members' and the customer's perspectives.

According to this semester timetable, this meeting ends the first iteration. The developed product is presented to the customer and feedback is given on the accomplishment of the requirements.

The presentation to the customer includes the acceptance tests of those customer stories developed during the first iteration, measures taken during the iteration, and presentations of the product from the perspective of the role holders.

Based on this presentation, the team receives feedback from the customer and facilitates a reflective session which focuses on the development process and product.

This meeting is also the beginning of the second iteration: students listen to the customer's priorities for the second iteration (as is described with respect to the first iteration in Chapter 3, Customers and Users).

7.5.4 Intermediate Course Review and Reflection

In addition to the focused reflective session that takes place as part of the studio learning, the end of the iteration is a good opportunity to facilitate additional

reflective processes that address also what has been studied so far in the course. This section presents several reflective activities that can be facilitated with the course participants. The activities can be conducted with the academic coaches, with the students, and with the instructor, based on the specific situation in which the course is taught.

If the instructor participates in these reflective activities, it enables him or her to hear the students' voices, communicate with the students, and become familiar with what they find interesting, how they understand the material taught so far, and what bothers them. Based of such interaction, the instructor can decide about the exact continuation of the course teaching.

The academic coaches who guide the students in their agile software development methods have the opportunity to deepen their understanding of the development process, including what goes on in between the weekly meetings. The academic coaches would also be able to understand what additional guidance the students need and what agile practices should be further emphasized.

The students, who should gain the most out of this reflective week, will have an opportunity to summarize what they have learnt so far, and to start the second course iteration based on this comprehensive picture and their improved understanding.

We divide the reflective tasks into four types. First, we present tasks that can be performed with any groups of students (not necessarily the teams which develop the software product in the studio). After these tasks are carried out in groups, a discussion, in which all the students who take the course participate, should be facilitated. In this discussion each group presents its messages to all the students.

The second set of activities should be conducted with the teams that develop the software in the studio.

The third set of activities can be performed with the teaching staff of the course (instructors and academic coaches).

The fourth set of activities can be performed individually by each of the course participants.

Each set of activities contains several suggestions. The facilitators of the reflective sessions can choose the activities that fit the specific teaching and learning environment, and of course change them and add new activities.

7.5.4.1 Group Activities

The following tasks can be performed in groups, not necessarily groups that form development teams. In some cases it is preferable to form groups with students

who belong to different teams in order to enable them to share the different experiences gained in each team.

- Summarize the main concepts learned so far in the course. Explain why these are the main concepts in your opinion.
- Discuss connections between the topics you mentioned in Question 1.
- Illustrate the importance of the concepts you just chose. Illustrations can be case studies, problems, success stories, failure stories, pictures, applications to other domains, patterns in software development processes, and more.
- Document the process the group goes through during the development of these illustrations. How can this process be characterized?
- For each topic addressed so far in the course, describe situations that have occurred in the studios. Compare the different stories. What is common to all of them? In what ways do they differ?
- Analyze each topic discussed so far in the course within the HOT—Human, Organizational, and Technological—analysis framework. What lessons can you derive from such an analysis?
- If the group is composed of students from different development teams, identify lessons that all the teams learned during the first iteration of project development in the studio.
- What topics would you like to learn in the next course iteration?
- Review the topics to be learned in the continuation of this course (look at the book's Table of Contents). With respect to each topic, suggest at least three questions for which you would like to get an answer when the topic is learned.
- Summarize the main lessons you learned during this reflective process.

7.5.4.2 Activities for the Development Teams

This set of activities should be facilitated with the teams that develop the software projects in the studio. If the academic coach wishes to guide this reflective discussion he or she should be very sensitive, avoiding imposing his or her opinion. The different reflective tasks can be facilitated also by the team members. Indeed, the facilitation of such reflective processes requires some experience; this practice, however, can serve as a good opportunity to start acquiring this skill. If the team decides to facilitate these activities on their own, it is recommended that the academic coach be present just to navigate the discussion if needed and to

improve his or her own understanding of the team's perspective on the development process. In any case, the information shared in this process should be used in the future only in a positive manner by all the participants (the students and the academic coach).

- Identify the three main activities performed by the team so far that influenced positively the team performances. What characterizes these activities? How is each of them connected to the HOT analysis scale?
- Identify the three main activities performed by the team that influenced negatively the team performances. What characterizes these activities? How is each of them connected to the HOT analysis scale?
- As a team, select one lesson to be delivered to the other teams in the course. This lesson should be derived directly from your analysis of your agile development process in the studio. What is this lesson? Why is it important? Did you consider additional lessons? Why did you choose this lesson?
- Analyze the team interaction with the customer. Identify positive aspects of this interaction and points that can be improved with respect to this communication channel.
- Analyze the team interaction with the academic coach. Identify positive aspects of this interaction and points that can be improved with respect to this communication channel.
- Analyze connections between the activities you conduct in the studio and the contents of the lectures. What connections should be strengthened, in your opinion?
- Summarize the main lessons you learned during this reflective process. How can they be applied in the continuation of the development process?

7.5.4.3 Activities for the Teaching Staff—Academic Coaches and Instructors

The third set of activities is directed to the teaching staff of the course—instructors and academic coaches. The goal of these activities is to increase the awareness of the teaching staff of the students' perspectives on the course—their understanding of the course in general and their development process in the studio in particular—and to navigate the course continuation according to this understanding.

It is important to dedicate a specific time for these activities. If time is dedicated for these activities, their importance is highlighted and the chances to enjoy their benefits increase.

- Identify the best activities the students performed in the studio during the development process of the first iteration. Why, in your opinion, were these activities performed well? Are these activities connected to the material presented in the lectures?
- Identify the activities the students performed badly in the studio during the development process of the first iteration. Why, in your opinion, were these activities performed badly? Were these activities connected to the material presented in the lectures? How can their accomplishment be improved?
- How did the students interact as a development team during the first iteration? Did they use the role scheme to improve communication? If so, how can this communication be further supported? If not, why?
- Identify conflicts that arose between team members. What were their sources? Were they solved? If so, how? If not, how can they be solved? What activities can be performed with the students to solve these conflicts?
- How did the students interact with the academic coaches during the first iteration? Can this communication be improved?
- Identify the main characteristics of the students' accomplishments as a team. Analyze them within the HOT—Human, Organizational, and Technological—analysis framework.

7.5.4.4 Reflective Tasks—Personal Work

The following activities can be performed individually by each of the course participants—students, coaches, and instructors.

- What was the most influential event for you so far in the course? Why do you characterize this event as the most influential? How is this event connected to the HOT—Human, Organizational, and Technological—analysis framework of software engineering?
- Summarize the main lessons you learned during the reflective process you conducted in teams and individually.

7.6 Summary and Reflective Questions

1. What characteristics of agile development environments support learning processes?
2. Review the main agile software development methods. How does each of them support learning processes?
3. Assess your presentation to the customer.
4. What lessons did you learn from the team's accomplishments in the first iteration?
5. How did the reflective activities improve your understanding of the course contents?

7.7 Summary

In this chapter we focused on learning—a central element of software projects. We examined how the agile practice of short releases supports learning processes in general, and software development processes and the course learning in particular. This examination has been conduced from the constructivist perspective. In line with this perspective on learning processes, after the rationale for this practice is explained from the cognitive perspective, the learning community is invited to dedicate some period of time to its knowledge construction, to examine what has been learned so far in the course, and to think about the continuation of the course.

References

Derby E, Larsen D, Schwaber K (2006) Agile retrospectives: making good teams great. Pragmatic Bookshelf

Dijkstra EW (1972) Notes on structured programming. In: Dahl OJ, Hoare CAR, and Dijkstra EW (eds) Structured programming. Academic Press, New York

Eckstein J (2004) Agile software development in the large—diving into the deep. Dorset House Publishing, New York

Hamlet D, Maybee J (2001) The engineering of software. Addison Wesley, Reading, MA

Hazzan O (2002) The reflective practitioner perspective in software engineering education. J Syst Software 63(3):161–171

Hazzan O, Tomayko J (2003) The reflective practitioner perspective in eXtreme programming. Proceedings of the XP agile universe 2003, New Orleans, Louisiana, USA, pp 51–61

Kerth N (2001) Project retrospective. Dorest House Publishing

Leron U, Hazzan O (1997) Computers and applied constructivism. IFIP WG g.1. Information and communications technologies in school mathematics. Working conference—secondary school mathematics in the world of communication technologies: learning, teaching and the curriculum. Grenoble, France, pp 195–203

Piaget J (1977) Problems of equilibration. In: Appel MH, Goldberg LS (1977) Topics in cognitive development, volume 1: equilibration: theory, research and application. Plenum Press, New York, pp 3–13

Schön DA (1983) The reflective practitioner. BasicBooks

Schön DA (1987) Educating the reflective practitioner: towards a new design for teaching and learning in the profession. Jossey-Bass, San Francisco

8
Abstraction

Abstract

Software development is a complex task. Abstraction is one means used for reducing the complexity involved in software product development. One way by which abstraction is expressed is by removing details in order to simplify and capture a concept, finding a common denominator for generalization. Though abstraction is a useful tool, it is not always used; sometimes it is just difficult to think abstractly, and sometimes abstraction is not utilized due to a lack of awareness of its significance and its potential contribution. This chapter describes how abstraction is expressed in agile software development environments. Specifically, software design and architecture are abstractions used in this chapter to discuss the concepts of simple design and refactoring. In addition, we revisit subjects that have been introduced in earlier chapters of the book and analyze them from the perspective of abstraction.

8.1 Overview

Abstraction is a central issue in mathematics, computer science, and software engineering (Devlin 2003, Hazzan 1999, Kramer 2007). It is a cognitive means by which, in order to overcome complexity at a specific stage of a problem solving situation, we concentrate on the essential features of our subject of thought, and ignore irrelevant details. Abstraction is especially important in solving complex problems, as it enables the problem solver to think in terms of conceptual ideas

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_8, © Springer-Verlag London Limited 2008

rather than in terms of their details. The discipline of software engineering, which leans on the sciences of mathematics and computer science, uses abstraction for the accomplishment of the complex task of software development.

There are different situations in which practitioners who take part in software development processes, are required to think abstractly. For example, when listening to customer stories, teammates are sometimes exposed only to the details and should think more abstractly in order to construct a structural and more global meaning; discussions that take place when the design implications of the customer stories are pondered increase the level of abstraction.

Further, since abstraction can be addressed on different levels, moving between different levels of abstraction can help in problem solving situations. For example, during discussions about the software design, teammates should sometimes ask the customer clarification questions in order to improve their understanding of the details of a specific story. They talk to the customer, explain their concerns, and ask for elaboration. The customer tells the story again, relating to the different concerns raised. Based on the customer's explanations, the software design is clarified. This short description illustrates how moving from a global view of the system, i.e., a high level of abstraction, to a local and detailed view of the system, i.e., a low level of abstraction, and vice versa, improves the understanding of the customer's stories and the implied design.

Obviously, there are intermediate abstraction levels that can be used during the process of software development. However, the knowledge of how and when to move between different levels of abstraction does not always come naturally, and requires some degree of awareness. For example, a team member may remain in an inappropriate level of abstraction for too long a time, while the problem could be solved immediately if it were viewed on a different level of abstraction (Hazzan and Kramer 2007).

In this chapter we deepen the discussion about abstraction as it is manifested in agile software development environments. We do that by discussing the practices of design and refactoring and by revisiting, this time from the perspective of abstraction, some of the agile practices we have already met in the book.

8.2 Objectives

- Readers will become familiar with the concept of abstraction and with what the ability to abstract means.
- Readers will get acquainted with how abstraction is related to and manifested in agile software development environments.

- Readers will learn how the practices of simple design and of refactoring increase abstract thinking.
- Readers will become aware of how abstraction is expressed by different agile practices.

8.3 Study Questions

1. What is abstraction? How is abstraction expressed in software engineering?
2. Is abstraction important in software engineering? Explain your answer.
3. How do the following statements, expressed by practitioners, reflect the need to move between levels of abstraction?
   - "I need to gain a global view of the application in order to know how this method fits into this design."
   - "I truly believe that if I had a minute to think about these two objects, I'd have come up with the conclusion that they could be extracted into one class. But I must move on to the next development task."
   - "I need some time to think about the code without being swamped with all the details. I'm almost sure that if I could leave now and go to swim, I could come up with a solution. But I must stay late, like all the other members of my team."
   - "I wish I could join the programmers when they write the code. You ask why? I'm not sure if this design can be easily implemented in Java or C#."
4. Present at least three teammates' statements that reflect the need to move between levels of abstraction. How did you choose them? How would you move between the levels of abstraction in the statements you just described? For what purposes?
5. Choose at least three agile practices that you have learned so far in this book. Analyze them from the perspective of abstraction, addressing questions such as: Does this practice lead me to think in terms of different abstraction levels? Are advantages gained from thinking on different abstraction levels? Are there situations in which this shift between abstraction levels interrupts? Illustrate your claims with specific illustrations taken from software development processes.

8.4 Abstraction Levels in Agile Software Development

Abstract thinking is needed in software engineering in order to understand the requirements and simplify the product design, keeping it maintainable and ready for ongoing changes. Several agile practices make use of abstraction in general and the shift between different abstraction levels in particular. In what follows (based also on Hazzan and Dubinsky 2003) we illustrate the abstraction notion by different situations in which agile teams use abstraction either explicitly or implicitly.

8.4.1 Roles in Agile Teams

The role scheme described in Chapter 2, Teamwork, can be viewed as a means that encourages software developers to look, think, and examine the development process on different abstraction levels. More specifically, if a team member wishes to perform the personal role successfully, that is, to lead the project in the direction that the role specifies, he or she must gain a more global and abstract view of the developed application as well as of the development process. When working on a specific development task, the developer works on a lower abstraction level. Thus, the role holder has two mental images of the project: one includes the details of a specific task of the development process, and the other encompasses a global view of a certain aspect related to the entire project. These two perspectives improve the role holder's understanding of both the product and process, mutually support and complement each other, and further, promote abstract thinking.

For an illustration, we examine the tracker role. The tracker collects data according to a set of measures used by the team, e.g., the actual development time of specific tasks. The minimal set of measures usually includes information about the project's progress relative to previous estimations and a measure that deals with the product quality—for example, code coverage (see Chapter 5, Measures). While the data is being collected, the tracker communicates with the other teammates. The tracker also analyzes and presents the measures at every iteration. This analysis is carried out at a higher level of abstraction than that needed for the actual data gathering. The former examines the project from a global view; the latter delves into the details of specific tasks.

Task

Choose at least two roles you have performed in software projects. Analyze them from the perspective of abstraction. What levels of abstraction were expressed

while performing each role? What characterized each level of abstraction? Did the performance of these roles encourage shifts between abstraction levels? How?

8.4.2 Case Study 8.1. Abstraction During Iteration Planning

The concept of short releases/iterations, and the planning sessions that direct the development process, encourage all the project stakeholders to move between levels of abstraction and to improve their understanding of the developed software gradually and periodically. While the release planning sessions inspire a global view of the developed product at a higher abstraction level, planning is conducted on a lower level of abstraction in the iteration planning sessions, considering the development tasks for the next iteration and their time estimation.

Task

Describe a scenario taken from a planning session in which you participated as part of a software project. Analyze the scenario from the perspective of abstraction. Can you identify abstract thinking? Can you indicate shifts between abstraction levels?

In what follows, based on Dubinsky et al. (2005), we analyze, from the perspective of abstraction, data taken from a planning session that took place as part of an agile introduction workshop (see Chapter 12, Change) for a software development team.

The following feedback was received from the participants after the first planning session had been conducted. This was the first time they had participated in such a session. The participants were asked to reflect on the contribution of the planning session to their understanding of the project and to describe the advantages as well as the disadvantages of this practice. Nine participants (out of ten), among them the project manager who played the customer role, reported that the planning session improved their understanding of the essence of the project and its requirements. Only one participant reported that the planning session did not contribute at all to his understanding.

Following is a sample of participants' expressions when describing the *advantages* of their first planning session: "Sharing information among teammates. Everyone knows what happens. Requirements understanding by all teammates. Acquaintance with all factors involved."; ". . .enables distribution to sub-tasks."; "The process was quick and enabled making many decisions in short time."; "The customer was available and understand our constraints."; We saw everything in

front of our eyes."; "Collaboration and improved acquaintance with the people we work with"

The focus on the advantages of the planning session, while neglecting its details, guided the participants to think on a higher level of abstraction than that on which they think during the planning session itself or during development activities. Therefore, they can observe more global ideas related to the customer, the project, and the work process.

Task

Describe how abstraction is expressed in several of the above teammates' expressions about the advantages of the planning session.

Following are participants' expressions when describing the *disadvantages* of their first planning session: "Some discussions were not necessarily related to the problem."; "Too many people are involved, lack of focus."; "Tendency to distraction. The customer is in pressure and doesn't remember everything."; "We didn't check duplications"; "We designed top-down and didn't verify our decision by bottom-up design."; "Sometimes, over talking caused lack of interest and concentration."; "Someone should manage the planning session so it won't be scattered."; "I didn't figure out how this will work. The planning seems very optimistic."

Task

Describe how abstraction is expressed in several of the above teammates' expressions about the disadvantages of the planning session.

The above feedback had been given before the team actually started developing by using an agile process. They indicate how the teammates grasped the planning session's benefits while still being suspicious and judgmental.

During the same reflective session the participants were also asked to describe their personal role in the team and how the planning session could help them perform their role (see Chapter 2, Teamwork). Table 8.1 presents the participants' feedback by their roles.

The participants' expressions from the personal role perspective reflect a higher level of abstraction than the level expressed when delving into the details of a specific development task. In general, the role scheme encourages moving between abstraction levels; the role perspective fosters thinking on a higher abstraction level than that on which teammates think in actual code development processes.

Table 8.1 Planning session reflections by roles (With kind permission of Springer Science and Business Media).

Role	How can the planning session help you with performing your role?
Coach	"–Tasks distribution. –Accountability of all teammates. –Statements regarding times. –Working according to a method. –Coordination of expectations."
Tracker	"Getting acquainted with all tasks and time estimations to be used for control."
Customer	"Receiving a realistic picture regarding my requirements—what is possible and when."
Acceptance tester	"The planning session will help me realize the size and scope of the tests."
Continuous integrator	"Keeping communication with the customer. Presenting results to the customer, sharing development constraints with the customer."
Designer	"Enables figuring out the structure of the system and influencing it, since all tasks can be easily observed."
In charge of infrastructure (an additional role that was required in this project)	"The planning session can help me plan how the work and development environment should be arranged for the project, which tools will be used."
Unit tester	"Getting acquainted with all tasks and understanding functionality before implementation."
Presenter and documenter	"Understanding the processes in the project."
Code reviewer	"Can help in thinking of possible ways to code."

Task

Illustrate the statement that "the role perspective fosters thinking on a higher abstraction level than that on which teammates think in actual code development processes" by at least two expressions of different role holders.

8.4.3 The Stand-Up Meeting

The stand-up meeting is conducted at the beginning of every development day (see Chapter 4, Time). It takes about ten minutes, in which each teammate describes in up to one minute what he or she accomplished the day before with respect to project development, what he or she is going to perform today, and the main problems encountered, if any. The meeting goal is to share relevant information about the project and to launch the development day. The coach who listens to the main problems can take care of them briefly during the stand-up meeting or, if needed, during the development day. The teammates stand during the meeting to make it short and concise.

Tasks

1. Stand up and perform an individual stand-up meeting that relates to a specific topic in your life. State in up to one minute what you did yesterday with respect to the said topic, what you are going to do today, and the main problems encountered, if any.

   Analyze your stand-up meeting from the perspective of abstraction. What levels of abstraction did you move between? When did the move/s happen? Why?

2. Perform a stand-up meeting with your team. Analyze it in the same way you analyzed the individual stand-up meeting.

The idea of a stand-up meeting is also used in the "scrum of scrums" practice. In this practice, representatives from all the teams in the project, who are in charge of the development process in their teams, conduct a stand-up meeting based on the same elements as a team stand-up meeting. That is, each representative describes what his or her team performed yesterday with respect to project development, what his or her team is going to perform today, and the team's main problems, if any. The need to summarize the team activities requires the team representative to take a more global and abstract view than the local detailed view needed during the team stand-up meeting, in which the team representative speaks as a teammate.

8.4.4 Design and Refactoring

Software design is carried out on a higher level of abstraction than code development. In other words, the development of a specific part of the software includes details which are eliminated when the design is constructed. Design is used to simplify communication with respect to the software and to represent the structure of the software components in an organized manner that can be understood by the different developers involved. Design can be sketched by some visual model and should be kept simple and clear.

Refactoring (Beck 2000, Fowler 1999, Highsmith 2002) or redesign means that we improve the software design without adding functionality. Refactoring is based on the current design and it attempts to simplify it and ease future changes. This activity requires thinking at a higher abstraction level than the level of abstraction needed when dealing with the design itself.

Reaching a simple design is not a simple task, and therefore refactoring is one of the practices that people find hard to accomplish. This difficulty can be

explained by the need to think about the code at a higher level of abstraction than the level of abstraction on which the code was written.

Since the practice of refactoring encourages programmers to keep improving code structure and readability without adding functionality to the code, the essence of refactoring is a gradual process of code improvement. More specifically, and especially for cases in which the final "correct" structure of the code and design cannot be predicted in advance, refactoring serves as a tool that leads and supports the team members in a gradual process of code and design improvement.

The inclusion and legitimization of refactoring as part of the development method delivers a clear message—that it is acceptable to stop the development of new tasks from time to time and to allocate time for code improvement. This improvement, in turn, eases future development.

Further, in practice, when a need for an extensive refactoring is acknowledged and is agreed to by the customer in the planning session, time is allocated for refactoring, and the same activities conducted with respect to code development, such as breaking down the refactoring activity into small parts and time estimation, are conducted with respect to code refactoring.

Ongoing refactoring takes time. The return on this investment, however, is expressed in maintenance activities, in which changes are inserted in clear and easy-to-change code.

Tasks

1. How is refactoring related to learning processes as described in Chapter 7, Learning?
2. Describe a situation is which you carried out a refactoring process. Reflect on this process by answering the following questions:
   - What were the reasons for the refactoring? How did the need for refactoring emerge?
   - How did you carry out the refactoring activity?
   - What difficulties did you face during this process?
   - Compare the situation before you started the refactoring and when you ended it. In what ways, if at all, did you achieve your goals? Are there differences between the two situations? What are they? Did these differences change your work with the code in later stages?
   - Did the refactoring improve your understanding of the code structure? If so, in what ways?

   - Predict in what ways the changes you made during the refactoring process might influence someone who did not write the code but might add to it a specific functionality.

3. Kent Beck (Fowler 1999) says that refactoring "is like a new kind of relationship with your program. When you really understand refactoring, the design of the system is as fluid and plastic and moldable to you as the individual characters in a source code file. You can feel the whole design at once. You can see how it might flex and change—a little this way and this is possible, a little that way and that is possible" (333). How do you conceive refactoring? What does Beck's description mean to you? Can you describe a specific situation that demonstrates Beck's assertion?

4. Sometimes people tend not to refactor. Can you explain why?

5. Sometimes refactoring raises resistance. Can you predict why?

6. Opdyke (Fowler 1999) explains why developers are reluctant to refactor (313). For each of the following statements, explain its source and indicate whether you agree with it or not:

   - Refactoring should be executed when the code runs and all the tests pass. It seems that refactoring wastes time.
   - If the benefits of refactoring are long-term, why exert the effort now? In the long term, developers might not be with the project to reap the benefits.
   - Developers might not understand how to refactor.
   - Refactoring might break the existing program.

7. If someone in your team put forth one of the following claims regarding refactoring, how would you answer him or her?

   - "I'm paid to write new, revenue-generating features."
   - "Refactoring is an overhead activity."
   - "It is hard for me to see the benefits of refactoring."

8.5 Abstraction in Learning Environments

The eighth meeting in the Studio is dedicated to the beginning of the second iteration. In the previous meeting, the product, which was developed during the first iteration, was presented to the customer, and the teammates listened

to the customer feedback. Also, new stories for the second iteration were told by the customer and the teammates were required to arrive at this eighth meeting with a high level design for the new stories and a suggested list of development tasks that fit these stories.

During this meeting, the planning session is completed by preparing a list of tasks that are split among the teammates according to their available time during this iteration. Load balance is also performed (see Chapter 4, Time). If the academic coach concludes that the team can perform the entire planning session on their own, as a self-organized team, he or she can leave the studio and let the team complete the planning session and publish the results—a list of personal task ownerships—in the electronic forum. At the end of this meeting, the second iteration of the product development starts.

8.5.1 Teaching and Learning Principles

The following teaching and learning principle deals with awareness of abstraction levels. In our list of teaching and learning principles presented in Chapter 14, Delivery and Cyclicality, this principle is number 9.

8.5.1.1 Teaching and Learning Principle 9: Be Aware of Abstraction Levels

During the process of software development, teammates are required to think at different abstraction levels and to move between abstraction levels.

Accordingly, this teaching principle suggests that the instructor and the academic coach should be aware of the abstraction level at which each stage of each lesson/activity is performed. Based on this awareness, they should then decide whether to stay at this abstraction level, or, alternatively, whether there is a need to guide the learners to think in terms of a different level of abstraction. It is further suggested that they explicitly highlight the movement between abstraction levels and discuss with the learners the advantages, as well as the disadvantages, of such moves (Hazzan and Kramer 2007).

The iteration-based course structure supports this principle. It allows the learning community to start, in the first stages of the course, with a more specific examination of the main course ideas, and to proceed to a more abstract perspective in the later stages, based on the understanding gained in the earlier stages.

8.5.2 Case Study 8.2. RefactoringActivity

This case study describes a refactoring activity facilitated with 32 students who developed software products as part of two project-based courses taught in two different academic institutions. The academic coaches of both courses attended the activity session. The subject was refactoring and the goal was to increase the students' awareness of abstraction as well as their abstraction skills. The students worked on the activity in eight mixed groups, formed to increase diversity (see Chapter 9, Trust).

The hands-on section is based on three worksheets. The first worksheet defines the practice of refactoring and its goals, and, based on that definition, asks the students to suggest examples of refactoring operations. Then two code excerpts are presented, and the students are asked to suggest refactoring operations for each.

Following is one of the code examples: [1]

```
SimpleString listHead = SimpleStringLinkedList(head);
counter = objectCounter(listHead);
listHead = mergeSort(listHead,counter);
listHead = SimpleStringCleanUp(listHead);
counter = objectCounter(listHead);
String[] cityArray = new String[counter];
createCityArray(listHead,cityArray,counter);
int[][] distanceArray = new int[counter][counter];
initializeDistanceArray(distanceArray,counter);
createDistanceArray(head,distanceArray,cityArray);
String[][] timeArray = new String[counter][counter];
initializeTimeArray(timeArray,counter);
createTimeArray(head,timeArray,cityArray);
```

Six groups suggested the following refactoring operations (two groups did not suggest any operation):

- There is a need to add documentation; there is a need to unify the three arrays.

[1] This code was taken from a refactoring example by Jaela Gulamhusein and Albert Choi (example by Michael Hanna). The URL is not accessible anymore.

- Add documentation and meaningful names for variables; listHead can be given by reference (&); distanceArray and timeArray will be implemented by a data structure whose implementation is not transparent to the user.
- We use only the number of elements in SimpleStringLinkedList; thus, performing the mergeSort is redundant; the two last parts do the same operations and can be unified into one function.
- Use a class/template to create and manage the arrays; inherit for city, distance, time.
- Create a new variable listHead1 for rows 3–14; create a class named CityArray and execute line 6 in the class constructor; the same for segments 3, 4.
- Create objects distance, time, city and enter them functions and fields; enter mergeSort under SimpleStr since this is an essential operation for it.

Task

What can be concluded from the fact that different student groups arrived at different suggestions for refactoring?

The second worksheet starts with a section that elaborates on the importance of refactoring. Then the refactored code for both examples is presented and students are asked to explain the refactoring operations.

The third worksheet describes cases in which refactoring is required and some refactoring operations are illustrated. Based on this description, students are asked to suggest refactoring operations for their current software project. All eight groups suggested refactoring operations for the two projects developed in the two institutions.

Finally, students are asked to answer three summary questions, presented in Table 8.2.

Tasks

1. Based on the students' answers presented in Table 8.2, what can you learn about thinking at different levels of abstraction?
2. Based on the students' answers presented in Table 8.2, what can you learn about the refactoring activity and its understanding?
3. Based on the students' answers presented in Table 8.2, what can you learn about the students' perceptions of refactoring?

Table 8.2 Reflections on refactoring

Group #	In which situations is it worth using refactoring?
1	Before the addition of new features
	When future reuse is expected
	When time limitations have caused quick and dirty writing
2	When functions become big
	When classes execute different actions, there is a need to split
3	When there is a long complex code and we would probably need to extend it
	When a project is divided into several parts and there is a chance that we will want to use these parts in the future
4	When we feel that the code becomes too complex
	When class functionality needs to use external modules
5	No answer
6	When there is code duplication
	When a method includes too much functionality
	When there is a complex code
7	When we receive new stories from the customer
	In the integration phase
8	When the code is not modular enough
	When the code isn't flexible for changes
Group #	**In your opinion, what is the connection between unit testing and refactoring?**
1	The unit testing should check that the refactoring did not interfere with the function
2	Makes the code more focused, prevents redundant code, simplifies the program
3	Unit tests are created in order to check that the system behaves correctly after refactoring
4	As the level of refactoring increases, the testing will be better, easier, and there will be less bugs and nonsense
5	After the refactoring there is a need to re-execute all the unit tests for all the units in the project
6	It is much easier to execute unit testing for small independent units that have unit functionality
7	During the testing, we can reveal that our design is not good enough (the tests will not pass in this design) and then ... refactoring will solve all our problems
8	After the refactoring we will execute the unit tests and will check that the change that was performed at a specific point in the system does not interfere with other points
Group #	**In your opinion, what is a reasonable frequency for refactoring operations?**
1	When code reuse is required
2	In every iteration as needed (and also during the planning)
3	In our opinion, if we use code standards, then not so frequent. But for our projects it is worthwhile to do it at the beginning of every iteration. The reason is the addition of more user stories
4	Once a week and every time you achieve a new understanding of the customer requirements

Table 8.2 (continued)

Group #	In which situations is it worth using refactoring?
5	Every iteration
6	It depends on the project progress. You need a reasonable amount of units, and you need to prevent a situation where the refactoring damages the program
7	At the end of every iteration or after a change in the customer requirements
8	At the beginning of every iteration to fit the new requirements

8.6 Summary and Reflective Questions

1. Choose at least three agile practices that were not addressed in this chapter and explain how they guide practitioners to move between abstraction levels during the development process of a software project.
2. Recall a situation in which you considered at different levels of abstraction a software project that you developed. Identify cases in which you felt the need to think at different levels of abstraction. Did that thinking process at different levels of abstraction help you? Did it interrupt you?
3. For each of the common activities conducted during a typical process of software development (planning, design, etc.), identify how it can be expressed at different levels of abstraction. Can it be characterized only at one level of abstraction? If so, how? If not, why?
4. Describe your cell phone (GPS, or any other tool) on at least four levels of abstraction. Outline the main considerations that guided you in choosing these descriptions. Had you been asked to develop a software system for this tool, how would the different descriptions influence the development process?
5. In your next software development session, reflect on your development process from the perspective of abstraction. Address questions such as: When do you move between abstraction levels? Do you increase the level of abstraction? Do you move to a lower level of abstraction? Why do you move to another level of abstraction? Do you change the object of your thoughts? Does this move support your thinking? If so, how? If not, why?
6. Imagine you are asked to develop a software system for a specific functionality. Describe two processes by which this system can be developed. The first process is constantly guided by abstraction; the second uses no abstraction. After completing the formulation of the two processes, compare them and draw your conclusions. Can you identify connections between these processes and agile software development?

7. Analyze the concept of abstraction and the practice of refactoring within the HOT—Human, Organizational, and Technological—analysis framework.

8.7 Summary

In this chapter we focus on abstraction and show how agile software development guides abstract thinking in general and the transition between abstraction levels in particular. One of the main messages of this chapter is that the shift between levels of abstraction increases teammates' understanding of both the development process and the product.

This chapter starts the second iteration of the book, as well as of the software product developed in the learning environment. In the learning section, we present a refactoring activity that usually requires thinking at a high abstraction level.

References

Beck K(2000) Extreme programming explained. Addison-Wesley, Reading, MA

Devlin K (2003) Why universities require computer science students to take math. Commun ACM 46(9):37–39

Dubinsky Y, Hazzan O, Keren A (2005) Introducing extreme programming into a software project at the Israeli Air Force. Proceedings of the 6th international conference on extreme programming and agile processes in software engineering, Sheffield University, UK

Fowler M (1999) Refactoring—improving the design of existing code. Addison-Wesley, Reading, MA

Hazzan O (1999) Reducing abstraction level when learning abstract algebra concepts. Educational studies in mathematics 40. Kluwer Academic, pp 71–90

Hazzan O, Dubinsky Y (2003) Bridging cognitive and social chasms in software development using extreme programming. Proceedings of the fourth international conference on eXtreme programming and agile processes in software engineering. Genova, Italy, pp 47–53

Hazzan O, Kramer J (2007) Abstraction in computer science & software engineering: a pedagogical perspective. Featured Frontier Columnist. System Design Frontier 4(1):6–14

Highsmith J (2002) Agile software development ecosystems. Addison Wesley, Reading, MA

Kramer J (2007) Is abstraction the key to computing? Commun ACM 50(4):37–42

9
Trust

Abstract

Software is an intangible product. Therefore, it is difficult to understand its development process using our regular senses; the exact status of the development process is not always clear; misunderstandings may emerge. Consequently, it is sometimes difficult to establish trustful professional relationships, and team members may tend not to trust each other. This chapter focuses on how agile software development fosters trust among team members. We first explain how agile software development makes the development process more transparent and thus makes the process and the developed product more understandable. Then, using the "prisoner's dilemma" framework taken from game theory, we explain why and how a transparent process does indeed foster trust among team members. Based on this working assumption, i.e., that the transparent nature of agile software development fosters trust among team members, we focus on how agile software development enhances ethical behavior and diversity in a way that improves process and product quality.

9.1 Overview

Since software engineering deals with the development of an intangible product—software—its development process cannot be managed by our regular senses—seeing, hearing, etc. Rather, other means must be employed for process control. In addition, software intangibility makes it difficult to identify each

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_9, © Springer-Verlag London Limited 2008

individual's contribution. Therefore, it might be difficult to foster trustful relationships in such an environment.

This chapter focuses on how trust is fostered by agile software development. The basic notion addressed in this chapter is how the agile software development environment makes the development process more transparent and sensible to the developers. Next, the focus is placed on conditions for cooperation, exploring how game theory helps explain why and how a transparent process increases trust and cooperation among team members.

Such a development environment, in which trustful relationships exist, enhances ethical behavior and diversity. The contribution of ethical behavior and diversity to the software development process and product is explained. Some limitations of diversity are explained as well.

The learning session of this chapter introduces the teaching and learning principle that advocates the establishment of diverse teams.

9.2 Objectives

- Readers will apprehend how agile teams can be empowered.
- Readers will deepen their understanding about the nature of agile software development in general and how it makes the development process more transparent in particular.
- Readers will become familiar with how agile software development can be analyzed from a game theory perspective.
- Readers will become familiar with the notion of ethics and how the agile software development environment supports ethical behavior.
- Readers will become familiar with the concept of diversity, understand its role in agile teams, and realize how diversity can enhance and support agile teams.

9.3 Study Questions

1. Explore the field of game theory. Suggest main ideas taken from game theory that can be utilized for the understanding of software development in general and agile software development in particular.
2. Discuss ways to increase trust among software team members. How do these ways enhance cooperation among team members? What characterizes these ways?

3. What is ethics? What professions have codes of ethics? Based on what ideas does a code of ethics rely? Can the software engineering community learn from these codes?
4. How can ethics improve the way a profession achieves its goal? How can ethics contribute to the creation of a community of practice?
5. How can ethics be expressed in software engineering processes? Illustrate your ideas with specific scenarios.
6. What is diversity? What connections can you find between diversity and software development in general and agile software development in particular?
7. Explore different ways by which your team can benefit from diversity.
8. Explore different ways by which your team can suffer from diversity.
9. Describe scenarios which illustrate how diversity may improve team performance.

9.4 Software Intangibility and Process Transparency

In previous chapters we mentioned connections between properties of software development processes and the fact that software is an intangible product. For example, in Chapter 6, Quality, it is suggested that since software is an intangible product, it requires a different development process in general, and a different approach towards the concept of software quality in particular, than do tangible products.

Since software is an intangible product, its development process is not transparent. In other words, when dealing with an intangible object, how can we know the exact development stage the process has reached, what has been accomplished by the teammates so far, what units or modules have already been tested, whether refactoring has been carried out, and so on? Therefore, in such environments it may be difficult to build trust.

In what follows, we show how several basic agile concepts increase project visibility, making the development process more transparent. This discussion is based on Hazzan (2007).

Whole team. This concept implies that all team members sit together in one space, including role holders that traditionally belong to separate teams (e.g., testers and designers). In agile development environments, the walls serve as a means of communication, constituting an informative workspace. Among

HOT

other items that the team decides on, the information posted on the walls includes the status of the personal tasks that belong to the current iteration and the measures taken. Thus all participants can see all that goes on all the time. See Chapter 1, Introduction to Agile Software Development, for further details about the collaborative workspace. In addition, the entire team holds daily stand-up meetings, which usually take place in the morning (see Chapter 4, Time). In these meetings, each team member presents the status of what he or she accomplished the day before, what he or she plans to do during the day to come, and problems he or she faces, if any.

Short releases. The actual detailed plan of the short releases and iterations is carried out during a planning session, in which *all* relevant parties participate—customer, team members, management representatives, and so on (see Chapter 3, Customers and Users). This activity, which usually takes a full day, includes a presentation of what was developed in the previous iteration, along with any relevant measures taken, and the planning for the next iteration. At the end of the day, a balanced workload is ensured among all team members. In the course of this day, a reflection process is also facilitated, in which development progress so far is analyzed and lessons learned (see Chapter 11, Reflection). The fact that all sides participate in this day, the nature of the activities that take place during the day, and the fact that it takes place every week or two, all increase process visibility and make the entire development process more transparent.

Time estimations. In agile software development, the teammate who is in charge of a specific development task also estimates the time needed for its development (see Chapter 4, Time). This increases the teammate's responsibility to perform well, and also enhances process transparency. The message conveyed is that all teammates know what each developer has committed to in terms of time estimations.

Measures. Measures are an essential element of agile software development (see Chapter 5, Measures). With respect to the discussion in this chapter, this means that visible measures are used to increase the transparency of the development process.

Customer involvement. In agile software development, all team members have access to the customer during the entire development process. This is particularly true during the planning game, in which all team members communicate with the customer, as mentioned earlier (see Chapter 3, Customers and Users). This direct communication channel enhances both process transparency and the chances that the software requirements are communicated correctly.

Testing. Testing is an integral part of an agile software development process (see Chapter 6, Quality); it ensures a more transparent process, because it clarifies who is in charge of the testing of each developed unit. Furthermore, acceptance

tests, which are defined by the customer and outline how each functionality should be tested, clarify the requirements and lead to a more transparent process.

Pair programming. Pair programming implies that all team members become familiar with all parts of the software, and thus process transparency is increased (see Chapter 1, Introduction to Agile Software Development).

Tasks

1. For each of the above agile practices explain how the fact that it increases process transparency might foster trust among team members.
2. Within the HOT—Human, Organizational, and Technological—analysis framework, suggest connections between the fact that software is an intangible product and its development process.

9.5 Game Theory Perspective in Software Development

In Chapter 2, Teamwork, we examined reward allocation among software teammates. In this chapter we continue this discussion and examine the issue from a game theory perspective. Specifically, we explain how the fact that agile methods make development more transparent increases trust, and consequently, cooperation, between team members is enhanced.

Game theory is concerned with the ways in which individuals make decisions, where such decisions are interdependent. For this purpose game theory uses theoretical fields such as mathematics, economics, and other social and behavioral sciences. The word "game" indicates the fact that game theory examines situations in which participants wish to maximize their profit by choosing particular courses of action, and in which each player's final profit depends on the courses of action chosen by the other players as well.

The "prisoner's dilemma" is a game theory framework that illustrates how a lack of trust leads people to *compete* with one another, even in situations in which they might gain more from cooperation. We will show that the transparency of agile software development eliminates the basic condition of the prisoner's dilemma and thus increases trust. The following analysis is based on Hazzan and Dubinsky (2005).

In the simplest form of the prisoner's dilemma, each of two players can choose, at every turn, between two actions: cooperation and competition. The working assumption is that none of the players knows how the other player will behave and that the players are unable to communicate. Based on the choices of the two

players, each player gains points according to the payoff matrix presented in Table 9.1, which describes the game from Player A's perspective. A similar table, describing the prisoner's dilemma from Player B's perspective, can easily be constructed by replacing the locations of the values 10 and (–10) in Table 9.1.

The values presented in Table 9.1 are illustrative. They do, however, indicate the relative benefits gained from each choice. Specifically, it can be seen from Table 9.1 that when a player does not know how the other player will behave, it is advisable for him or her to compete, regardless of the opponent's behavior. In other words, if Player A does not know how Player B will behave, then in either case (whether Player B competes or cooperates), Player A will do better to compete. According to this analysis, both players will choose to compete. However, as can be seen, if both players choose the same behavior, they will benefit more if they both cooperate rather than if they both compete.

The prisoner's dilemma is manifested in real life situations in which people tend to compete instead of to cooperate, although they can benefit more from cooperation. The fact that people tend not to cooperate is explained by their concern that their cooperation will not be reciprocated, in which case they will lose even more.

The dilemma itself stems from the fact that the partner's behavior (cooperation or competition) is an unknown factor. Since it is unknown, the individual does not trust the other partner, nor does the partner trust the individual, and, as described in Table 9.1, both parties choose to compete. We emphasize that this behavior is common to all human beings in situations in which cooperation can not be ensured.

Tasks

1. Find on the Web a Java applet that illustrates the prisoner's dilemma and play with it. What was your strategy? Did you win? What did you learn from this game?
2. What connections can you find between the fact that software is an intangible product and the prisoner's dilemma?
3. What connections can you find between the prisoner's dilemma framework and software development in general and agile software development in particular?

We now use the prisoner's dilemma to analyze software development environments. It should be noted first that, in these environments, cooperation (and

Table 9.1 The prisoner's dilemma from player A's perspective (with kind permission of Springer Science and Business Media.)

	B cooperates	B competes
A cooperates	+5	−10
A competes	+10	−5

competition) can be expressed in different ways, such as information sharing (or hiding), using (or ignoring) coding standards, clear and simple (or complex and tricky) code writing, etc. It is reasonable to assume that expressions of cooperation will increase the project's chances of success, while expressions of competition may add problems to the process of software development.

Since cooperation is so vital in software engineering, it seems that the quandary raised by the prisoner's dilemma is even stronger in software development environments. To illustrate this, let us examine the following scenario, according to which a software team is promised that if it completes a project on time, a bonus will be distributed among the team members according to the individual contribution of each team member to the project. In order to simplify the story, we will assume that the team comprises only two members—A and B. Table 9.2 presents the payoff table for this case.

The main difference between Table 9.2 and the original prisoner's dilemma table (Table 9.1) lies in the cell in which the two players compete. In the original table, this cell reflects an outcome that is better for both players than the situation in which the player cooperates and the opponent competes. In software development situations (Table 9.2), the competition-competition situation is worst for both team members. This is explained by the vital need for cooperation in software development. It can be seen from Table 9.2 that in software development environments, partial cooperation (reflected in Table 9.2 by the cooperation of only one team member) is preferable to no cooperation at all.

In general, software team members are asked to cooperate. At the same time, however, if the development process is not transparent, they are unable to ensure

Table 9.2 The prisoner's dilemma in software teams (With kind permission of Springer Science and Business Media.)

	B cooperates	B competes
A cooperates	The project is completed on time. A and B get the bonus. Their personal contribution is evaluated as equal and they share the bonus equally: 50% each	A's cooperation leads to the project's completion on time and the team gets the bonus. However, since A dedicated part of his or her time to understanding the complex code written by B, while B continued working on his or her development tasks, A's contribution to the project is evaluated as less than B's. As a result, B gets 70% of the bonus and A gets only 30%
A competes	The analysis is similar to that presented in the cell "A cooperates/B competes." In this case, however, the allocation is reversed: A gets 70% of the bonus and B gets 30%	Since both A and B exhibit competitive behavior, they do not complete the project on time, the project fails and they receive no bonus: 0% each

that their cooperation will be reciprocated. In such cases, even if there is a desire to cooperate, as indicated by the prisoner's dilemma Table 9.1, each team member will prefer to compete. However, as indicated by Table 9.2, in software development situations such behavior (expressed by the competition-competition cell) results in the worst result for both team members.

In what follows we illustrate, by the agile practices of test-driven development (see Chapter 6, Quality) and refactoring (see Chapter 8, Abstraction), how agile software development enhances trust among team members. This is because these practices, among others, make the development process transparent and eliminate the working assumption of the prisoner's dilemma that the way the other teammates will behave is unknown. Thus, within such an environment, teammates are led into cooperation-cooperation situations. In other words, we explain how, from a game theory perspective, agile practices lead to the establishment of development environments whose atmosphere can be characterized by the cooperation-cooperation cell of the prisoner's dilemma.


HOT

Test-Driven Development. The meaning of cooperation in the case of test-driven development is that all team members verify that their code is tested and does not fail; competition means that team members do not verify that the code is fully tested.

Since agile development is the development environment, all team members are committed to apply its practices and, in particular, the practice of test-driven development. Specifically, this means that all team members are committed to verify that their code is fully tested. In other words, it implies that all team members cooperate and apply this practice. Thus, the unknown behavior of the others, which is the source of the prisoner's dilemma, ceases to exist. Consequently, team members face no (prisoner's) dilemma whether to cooperate or not, and since they are guided by the practice of test-driven development, they all cooperate and test their code with no concern about whether their cooperation will be reciprocated or not. Since, for purposes of software quality, it is required that software be intensively tested, all team members benefit more from this practice than if they had chosen to be in competition with one another. Thus, the practice of test-driven development yields a better outcome for all team members.


HOT

Refactoring. The meaning of cooperation in the case of refactoring is that all team members make sure to stop their development tasks from time to time and improve their existing code readability and clarity (remember that time is allocated in the planning sessions for big refactoring activities); competition means that the team members do not pay attention to code readability and improvement and do not invest the time needed for refactoring.

In agile development, all team members are committed to apply agile practices and, in particular, the practice of refactoring. Specifically, this

means that all team members are committed to refactor the code when it is needed. In other words, since all team members are committed to working according to the agile development process, part of which is refactoring, they all cooperate, and apply this practice. Thus, the unknown behavior of the others, which is the source of the prisoner's dilemma, ceases to exist. Consequently, team members face no (prisoner's) dilemma whether to cooperate or not, and since they are guided by the practice of refactoring, they all cooperate and refactor their code when needed with no concern about whether their cooperation will be reciprocated or not. Since, for purposes of software quality and maintainability, it is required that software be refactored when needed, all team members benefit more from this practice than if they had chosen to be in competition with one another. Thus, the practice of refactoring yields a better outcome for all team members.

Tasks

1. Apply the above analysis to other agile practices. Which practices did you decide to start your analysis with? Why?
2. The above analysis of agile software development using the prisoner's dilemma is illustrated by practices that are directly related to code development—test-driven development and refactoring. Apply the above analysis to practices which are not code-related (though, of course, they may influence code quality).
3. There are variations and subtle points (and extensive literature) related to the prisoner's dilemma. Explore the relations of such variations and subtle points to agile software development.

9.6 Ethics in Agile Teams

Tasks

1. In your opinion, does the software engineering community need a code of ethics? If so, what situations are appropriate to be addressed by such a code? What situations should be not be addressed by a code of ethics?
2. In your opinion, what other occupations related to software engineering should have a code of ethics?

3. Suggest specific scenarios that illustrate the importance of ethics in software development environments in general and agile software development in particular. Are these scenarios different? Explain your answer.

Codes of ethics guide professionals how to behave in vague situations when it is not clear what is right and what is wrong. The need for a code of ethics arises from the fact that any profession generates situations that can neither be predicted nor answered uniformly by all members of the relevant professional community. In this section we examine how agile software development fosters ethical behavior by developers.

There are many ethical issues related to information technology, computing, and technology. To address this reality, the ACM/IEEE-CS Joint Task Force defined the Software Engineering Code of Ethics and Professional Practice (Version 5.2). Its short version is presented in what follows (for the full version look at http://info.acm.org/serving/se/code.htm).

The Software Engineering Code of Ethics and Professional Practice—Short Version[1]

Software Engineering Code of Ethics and Professional Practice
ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices
Short Version
PREAMBLE

The short version of the code summarizes aspirations at a high level of abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety, and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC—Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER—Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

[1] Copyright (c) 1999 by the Association for Computing Machinery, Inc. and the Institute for Electrical and Electronics Engineers, Inc. It is explicitly specified in the Code website that it may be published without permission as long as it is not changed in any way and carries the copyright notice.

3. PRODUCT—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT—Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES—Software engineers shall be fair to and supportive of their colleagues.
8. SELF—Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Naturally, all software development environments should encourage ethical behavior. In the context of this book we review several sections of the code of ethics from the perspective of agile software development. In general, the working assumption is that since agile software development processes are transparent, ethical behavior is encouraged in the same way as trust is fostered in agile software development (see the previous section). In particular, in what follows, we specify, with respect to several sections of the code of ethics, what agile practices support it.

2. CLIENT AND EMPLOYER—Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

This section of the code of ethics is fostered by the close interaction with the customer in agile software development (see Chapter 3, Customers and Users). Specifically, the fact that the customer is in close interaction with the team and the fact that all the project stakeholders hear the customers' requirements, further support the enhancement of this section of the code of ethics. The acceptance tests that are defined together with the customer further boost the ability to achieve ethical behavior in this respect.

3. PRODUCT—Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

Ethical behavior in this respect is achieved by several agile practices, such as measures (see Chapter 5, Measures), testing (see Chapter 6, Quality), and refactoring (see Chapter 8, Abstraction).

Task

How does each of the above mentioned agile practices, as well as other agile practices, reinforce ethical behavior with respect to the product?

4. JUDGMENT—Software engineers shall maintain integrity and independence in their professional judgment.

Integrity is maintained by the fact that agile team members are encouraged to raise problems they encounter, to discuss dilemmas, and to express their concerns. Several kinds of opportunities are provided to agile team members for dealing with these subjects, such as reflective and retrospective sessions (see Chapter 7, Learning, and Chapter 11, Reflection).

5. MANAGEMENT—Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

This section of the code of ethics is related to topics discussed in future chapters (Chapter 12, Change, and Chapter 13, Leadership). In these chapters, we will draw attention to this section of the code of ethics.

6. PROFESSION—Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

This section talks about the profession of software engineering. The standards that agile software development sets for the profession emphasize responsibility, accountability, fairness, and trust (see Chapter 2, Teamwork, and the discussion in this chapter).

7. COLLEAGUES—Software engineers shall be fair to and supportive of their colleagues.

This section of the code of ethics looks at the relationships among the individuals in a software development process and advocates fairness and support among them. This idea is fostered by the Agile Manifesto (see Chapter 1, Introduction to Agile Software Development) and is achieved, among other means, by the role scheme described in Chapter 2, Teamwork.

Tasks

1. How does the role scheme, described in Chapter 2, Teamwork, foster ethical behavior in general and with respect to Section 7 or the code of ethics?
2. How does agile software development foster ethical behavior with respect to Sections 1 and 8 of the code of ethics?

3. Describe scenarios which illustrate the importance of each section of the Code of Ethics of Software Engineering. Explain how the code of ethics guides us to cope with such situations.

In conclusion, we emphasize that by making the development process more transparent, agile software development supports ethical behavior. This is because a more transparent process means that behaviors are more visible; consequently, norms can be set and adhered to and ethical behavior is therefore more easily supported. As a result, trust and communication are increased among team members.

Tasks

1. Choose at least three agile practices and explore their contribution to ethical behavior in agile teams. What sections of the code of ethics do these practices support? Why did you select these practices?
2. Some communities of practice have a well known code of ethics (e.g., the code of medical ethics). What, in your opinion, is common to these codes of ethics and the Code of Ethics of Software Engineering?
3. Formulate a set of ethical behavior guidelines that your team members should adhere to.

9.7 Diversity

Diversity can be expressed in different contexts, such as nationalities, worldviews, genders, minorities, cultures, and life styles. Diversity can also be expressed with respect to internal characteristics, such as hobbies, skills, thinking styles, and interests.

In general, studies tell us that no matter how diversity is expressed, it benefits and enhances societies that foster it (e.g., Florida 2002). Diversity is also perceived as a powerful management practice (see, for example, Toyota's 21st Century Diversity Strategy[2] and Thomas 2004). This perspective is based on management theories that assert the added value of diversity (see the American Institute for Managing Diversity, http://aimd.org/).

At the same time, however, and mainly with respect to social and ethnic diversity, resistance is sometimes expressed towards diversity. The main argument presented is that people tend not to trust people who are not like them.

[2] Toyota's 21st Century Diversity Strategy: http://www.toyota.com/about/diversity/21stcenturyplan.pdf.

For example, Harvard professor Robert D. Putnam's research indicates that the effect of diversity is worse than had been imagined.[3] In diverse communities, Putnam's research shows, trust (even of one's own "race") is lower (Smith 2001, 2007). Also, according to Austin (1997) "there may be an optimal level of diversity that will stimulate creative thinking within a group, and the relationship between group diversity and creativity may be curvilinear" (342). Accordingly, Austin suggests that organizational members should be aware of the increased number of disagreements that may stem from increased diversity.

Diversity is introduced into agile development to exploit its benefit for software development. For illustration, we quote Kent Beck who presents diversity as an Extreme Programming principle in the 2nd edition of *Extreme Programming Explained*: "Teams need to bring together a variety of skills, attitudes, and perspectives to see problems and pitfalls, to think of multiple ways to solve problems, and to implement the solutions. Teams need diversity." (Beck with Andres 2005:29).

So far we have seen the transparent nature of agile software development and how it fosters cooperation and ethical norms. Within such conditions, diversity can also flourish. Though, as just mentioned, it is sometimes argued that diversity does not necessarily increase trust, it does depend on the atmosphere that a (working) environment inspires. This is because when trust is increased, team members are more open to new ideas and perspectives in particular and to diversity in general.

Task

1. Is your team diverse? If so, in what sense? Do you benefit from this diversity? If not, why? If so, how? If your team is not diverse, what is common to all the team members? Describe specific scenarios to illustrate your ideas.
2. How can trust support diversity? Describe possible situations that illustrate how trust and diversity mutually support each other.
3. Review how different software development methods express and foster diversity.

In what follows, we first explore how the agile approach enhances diversity. Then we examine how diversity can benefit agile teams and how diversity can improve the quality of software products.

[3] Look at http://www.amconmag.com/2007/2007˙01˙15/cover.html.

Diversity is enhanced in agile teams in different ways. First, the role scheme presented in Chapter 2, Teamwork, enables each team member to express his or her perspective on the development process and to influence it. When the project is scaled up, the role holder meetings enable the expression of different opinions based on the experience gained in each team. Second, the fact that all the project stakeholders participate in the planning session, as well as in the reflective sessions (see Chapter 11, Reflection), enhances the contribution and expression of different opinions, that may influence and contribute to the development process. Third, Beck says that "Diversity is expressed in the practice of Whole Team, where you bring together on the team people with different perspectives" (Beck with Andres 2005:29).

Teams may benefit from diversity. First, the more diverse a team is, the more diverse perspectives are elicited; consequently, teammates are exposed to other kinds of perspectives, and are able to use these different points of view in different situations. Second, the developed software product itself may be improved, because when different perspectives are expressed with respect to a specific aspect of its development, the chances that subtle issues will emerge are higher; consequently, additional factors are considered when decisions related to the developed product are taken. Third, the creation process is questioned more when diverse opinions are expressed, and, once again, we may get a more argument-based process on which specific decisions are based. Fourth, diversity reduces resistance to new ideas and establishes an atmosphere which is open towards alternative opinions. Finally, as more and more companies become global, diversity is becoming an integral characteristic of software development teams and, therefore, cannot be neglected (see Chapter 10, Globalization).

When different and diverse opinions are encouraged, the development process becomes more transparent, since it is clear how the different project stakeholders conceive of the different aspects of the development process. This openness to diversity naturally increases communication and, consequently, the development process is enhanced.

Tasks

1. Describe a scenario that illustrates how diversity benefits agile teamwork. What characterizes such scenarios?
2. Describe a scenario that illustrates how diversity may harm agile teamwork. What characterizes such scenarios?

3. Resistance is sometimes expressed towards diversity. These opinions sometime assert that diversity creates disorder. What is your viewpoint on such arguments? Explain and illustrate it.
4. How does the role scheme presented in Chapter 2, Teamwork, enhance diversity (beyond the way mentioned above)?
5. How does the reflection practice (see Chapter 11, Reflection) enhance diversity?
6. Suggest additional ways by which agile development processes enhance diversity. Do they influence the development process? If so, how?
7. Does diversity always empower agile teams? Try to present a balanced argument.
8. How does diversity influence the HOT—Human, Organizational, and Technological—aspects of software development processes in general and of agile software development in particular?
9. Choose at least three agile practices and explore their influence on diversity in agile teams.
10. In Chapter 10, Globalization, we mention diversity as one of the inherent characteristics of distributed teams. What, in your opinion, are the advantages and disadvantages of diversity in such situations?

9.8 Trust in Learning Environments

Meetings 9 and 10 are part of the development phase of the second iteration and are dedicated to the actual development. The academic coach can assess the project's progress and each student's involvement by participating in this development meeting, listening to the stand-up meeting, watching how students work on their development tasks and on their personal roles, talking to them, pair programming with them, and reading their reflections.

If needed, team discussions can be held in which students can raise specific issues for discussion.

9.8.1 Teaching and Learning Principle

We present one teaching and learning principle that deals with diversity. In our list of teaching and learning principles this is number 6. The complete list of teaching and learning practices appears in Chapter 14, Delivery and Cyclicality.

9.8.1.1 Teaching and Learning Principle 6: Establish Diverse Teams

Diversity can be expressed in different ways, such as nationalities, gender, minorities, cultures, life styles, and worldviews. The importance attributed to diversity is based on management theories that assert the added value of diversity. Therefore, and not surprisingly, diversity is perceived as a powerful management practice. In the same spirit, diversity is encouraged in agile software development.

This principle advocates the idea that teams should be diverse with respect to different factors, external (such as gender) and internal (such as perspectives). The more diverse a team is, the more diverse are the perspectives elicited, which in turn may improve software development and teammate communication.

9.9 Summary and Reflective Questions

1. Review a game theory website. Summarize five lessons you learned from this review. Suggest additional connections to, or implications of game theory for, software engineering in general and agile software development in particular.
2. How is trust expressed in your teamwork?
3. Describe situations in which cooperation between team members is enhanced when the rules of the development process are clear, transparent, and known to all.
4. Describe situations in which cooperation between team members is reduced when the rules of the development process are not clear, not transparent, and not known to all.
5. How can the role scheme presented in Chapter 2, Teamwork, enhance ethical behavior and diversity?
6. Describe situations related to your teamwork that illustrate the need for the Code of Ethics of Software Engineering.
7. Explain how diversity influences agile teamwork.
8. Discuss connections between the different ideas presented in this chapter.
9. Analyze the concepts presented in this chapter within the HOT—Human, Organizational, and Technological—analysis framework.

9.10 Summary

This chapter binds three concepts—cooperation, ethics, and diversity—under the notion of trust. We explain how agile software development increases trust by establishing a transparent development process. Based on this working assumption, i.e., that agile software development fosters a trustful atmosphere, we describe how cooperation, ethical behavior, and diversity can be fostered, flourish, and contribute to the development process and product.

References

Austin JR (1997) A cognitive framework for understanding demographic influences in groups. International J Organ Anal 5(4):342–359

Beck K, Andres C (2005) Extreme programming explained, 2nd ed. Addison Wesley, Reading, MA

Florida R (2002) The rise of the creative class. Basic Books

Hazzan O (2007) Agile software development and the nature of software development. Featured Frontier Columnist. System Design Frontier 4(3):28–32

Hazzan O, Dubinsky Y (2005) Social perspective of software development methods: the case of the prisoner dilemma and extreme programming. Proceedings of the sixth international conference on extreme programming and agile processes in software engineering. Sheffield University, UK, pp 74–81

Smith MK (2001, 2007) Robert Putnam: the encyclopedia of informal education. www.infed.org/thinkers/putnam.htm. Last update: November 05, 2007

Thomas D (2004) Diversity as strategy. Harvard Bus Rev 98–108 (http://www.gpworldwide.com/quick/sep2004/art2.asp)

10
Globalization

Abstract

In this chapter we address the concept of agility in a wider context. One topic on which we focus is globalization in terms of distributed teams; the second idea is the application of the agile approach for the management of non-software projects. Agile software development has evolved significantly during the last decade. In parallel to this evolution, globalization in software development has also emerged, and software is developed in many cases by teams which are spread across geographical areas, cultures, and nationalities. This reality, called global software development, has advantages as well as disadvantages. The most obvious advantage is the business aspect of cost reduction; the most problematic issues are communication and team synchronization. In this chapter we briefly describe the notion of global software development and explain how some agile practices help cope with the challenges involved. Specifically, we will see that the agile approach encourages a transparent global software development process. thus increasing information flow and project visibility and assistings in solving communication and synchronization problems. Further, the tightness of agile processes simplifies software project management. We also examine in this chapter the notion of agility beyond the software world and discover its usefulness in such projects

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_10, © Springer-Verlag London Limited 2008

10.1 Overview

Globalization is usually related to time, distance, and culture.

Referring to time, we cite Friedman's book *The World is Flat*: "... 'That's globalization,' said Nilekani. Above the screen there were eight clocks that pretty well summed up the Infosys workday: 24/7/365. The clocks were labeled US West, US East, GMT, India, Singapore, Hong Kong, Japan, Australia" (Friedman 2005:6). Chapter 4, Time, discusses the concept of time in general and with respect to the agile approach in particular. In this chapter we elaborate on its relation to global software development.

Referring to distance, a physical distance between teams which work together on the development of one software product increases the process complexity. It is claimed that even a fifty-meter distance can be considered a distributed development environment (Allen 1984 in Sangwan et al. 2007).

Referring to culture, this concept has been explored extensively with respect to different kinds and sizes of groups like nations, tribes, and teams. We define the concept of culture as a set of explicit and implicit norms, values, and beliefs, shared by the members of a group, which, on the one hand, influences directly the members' daily activities, behaviors, and interactions; and on the other hand, is shaped by these activities, behaviors, and interactions.

So far in this book we have addressed the concept of agility in the context of software projects that take place in software organizations. In this chapter, we address the concept of agility in a wider and more *global* context. One topic on which we focus is globalization in terms of distributed teams; the second idea is the application of the agile approach to the management of non-software projects, illustrating how that approach has been applied in the process of writing this book. In both cases, it will be seen that the agile approach offers solutions for similar challenges.

The readers are familiar with most of the notions we address, such as communication and short releases, since they have been presented in previous chapters of this book. Other notions, such as reflection and leadership, will be addressed in later chapters. In this chapter, however, all these notions are examined with respect to distributed teams and non-software projects, illustrating their fitness for such settings.

10.2 Objectives

- Readers will become familiar with the notion of globalization, how it is expressed in software development environments, and the challenges it introduces.

- Readers will get acquainted with how agile software development methods are implemented in global software development.
- Readers will learn how the agile culture fits into distributed software development environments.
- Readers will see how the agile approach can be used for non-software projects.
- Readers will realize how the book learning style itself reflects the agile approach.

10.3 Study Questions

1. What is globalization? Present at least three examples of how globalization is expressed in different disciplines.
2. What are the main advantages of global software development? Present evidence for your claims.
3. Present at least three problems that characterize global software development. For each problem present evidence and at least one solution.
4. Referring to the notion of diversity (see Chapter 9, Trust), explain how globalization of software development can promote diversity. How can diversity benefit the individuals? The team? The organization? How can diversity interfere?
5. Within what context are you familiar with the concept of culture? What are the culture's norms? How are these norms implemented on a daily basis?
6. Look at the proceedings of the agile conferences: What can be learned about the culture of the agile community? What values does it honor? Who plays a role in the agile culture? How is the culture of the agile community related to global software development?
7. In your opinion, how are the aspects of the HOT—Human, Organizational, and Technological—analysis framework expressed in global software development environments?
8. What agile ideas can be used for the accomplishment of non-software development tasks? Describe how they can be used.

10.4 The Agile Approach in Global Software Development

Global software development is employed when software companies set distributed teams to work together on product development (Carmel 1999, Herbsleb et al. 2001, Sahay et al. 2003, Sangwan et al. 2007). The motivation for global

development usually stems from the need to use the organization's resources cost-competitively, and the need to shorten time to market by around-the-clock development.

Some experience has already been gained with the implementation of agile software development in distributed environments (Sangwan et al. 2007). In this section, we show how the agile approach guides software development processes in global software development environments, focusing on the following topics: communication, planning, reflection, and organizational culture.

10.4.1 Communication in Distributed Agile Teams

Interactions among individuals who are part of the software development environment are emphasized by the Agile Manifesto (see Chapter 1, Introduction to Agile Software Development). Further, the application of a role Scheme in software teams enhances communication and interactions among teammates because of the teammates' accountability for cross-project activities (see Chapter 2, Teamwork).

Distributed teams cannot meet face-to-face and do not have "corridor meetings." Nevertheless, interaction between teammates is crucial for the success of distributed teams. Since the agile approach emphasizes the need for smooth and fruitful communication in distributed teams, communication issues should be constantly addressed by deciding, for example, on communication facilitators, communication channels, and communication measures.

In Sahay et al. (2003) three strategies for cross-cultural communication are suggested. The first is to minimize communication needs as much as possible; the second is to invest in learning the other side's culture and language; the third suggests assigning a person who serves as a bridge between the two sides, who understands the cultural as well as the technical issues of both sides. In Sangwan et al. (2007) tips are listed with respect to the facilitation of communication among distributed teams. It is suggested that the communication should be adequate, not too minor and not overwhelming, and, in any case, that it should be measured. In Carmel (1999), initial face-to-face meetings are suggested to enable smooth electronic communication in later stages.

The agile approach promotes communication for the development of a high quality product. In general, distributed teams adhere to the agile notion of communication by discussing how to exploit the benefits of communication, setting the resources and procedures needed for fruitful communication, and tracking the actual communication.

Tasks

1. How can the agile approach towards communication be implemented in distributed software teams? Suggest specific ways for such implementation.
2. Suggest specific measures of communication in distributed agile teams. What does each measure reflect? What values and norms does each measure foster?

10.4.2 Planning in Distributed Agile Projects

Planning is one of the tightest activities in agile software development. It is performed in an iterative manner—small releases of a few months each and short iterations of 2–4 weeks each (see Chapter 3, Customers and Users, and Chapter 4, Time).

Teams should be synchronized in order to develop a high quality product. In distributed teams the planning activity also serves coordination and synchronization purposes. Different techniques and tools are suggested for the planning of software projects in distributed environments. Cusick and Prasad (2006) present some recommendations that emerged from their experiences with many global non-agile development projects, that fit for implementation in agile software development environments as well. For example, "Limit phase durations to keep control. Shorter phases are easier to track and manage. Track all issues assiduously. Require interim deliveries to ensure quality."

Task

In your opinion, what problems led Cusick and Prasad (2006) to provide these recommendations?

10.4.3 Case Study 10.1. Tracking Agile Distributed Projects

This case study describes measures that were taken in a simulation of the agile distributed environment within the framework of an activity called "Planning with Distributed Teams," that was facilitated by Smits and Sulaiman in the Agile 2007 conference. The participants in this activity were practitioners involved in global software development and belonging to distributed teams.

During the simulation, two teams in two different time zones (Washington DC and India) performed a release planning session. In actuality, the two teams were in the same room, on two different sides, without direct eye contact, and

planned the release in accordance with a predefined requirement list. A project wiki was simulated by a flip chart that was handed over every "morning;" a conference call was simulated by sitting back to back in separate places in the room, so delays and communication freezes happened; finally, key roles were played by participants who came from different cultures.

When measuring the cost of this communication, both financial costs and human costs were considered and ranked between 1 (low cost) to 5 (high cost). Table 10.1 summarizes the measurements determined by each team, as expressed and ranked by them. As can be observed, people mainly referred to human cost.

After this experience, a person was assigned to be in charge of communication; some participants claimed that there was some improvement.

Tasks

1. Can you explain the selection of measures by the teams in the simulation?
2. What other measures would you suggest be used in this case?
3. Describe at least three characteristics of the communication facilitator.

10.4.4 Reflective Processes in Agile Distributed Teams

In Chapter 7, Learning, and Chapter 11, Reflection, we describe the reflection and retrospective activities which are essential for agile teams that wish to maintain and improve the development process. Reflection is also one of the most important tools to control and improve performances in distributed environments. It provides teammates with a way to talk about problems and discuss their main concerns. Further, it highlights information about the process and enables improvements to reduce some of the frustration that developers in distributed environments feel.

Table 10.1 Human cost in a distributed environment

Team A (DC) Cost description	Team A (DC) Rank	Team B (India) Cost description	Team B (India) Rank
We were stressed, didn't hear well	3	Cannot hear	4
Frustrations, cannot concentrate	5	Not engaged, nobody loves us	4
Misunderstanding, slow process with many interruptions	4	Do not feel engaged, don't know what the process is	4–5
		Frustration	4–5

In Nirenberg (2002) a leader in a global environment is described as a practitioner with self-reflection and social skills. These skills are needed for building a group consensus, taking into consideration personal and team characteristics.

Task

Suggest a retrospective activity for a distributed team in the middle of the first release of an agile project.

10.4.5 Organizational Culture and Agile Distributed Teams

The culture of a specific group is influenced by the culture of the nation as well as the organizational culture. Both are relevant for global environments. Connections between software development methods and cultural issues have been discussed previously (Yourdon 1997, Sawyer and Guinan 1998, Abrahmsson et al. 2002). For example, Highsmith (2002) says that "A particular culture is not necessarily change tolerant or change resistant—but it may resist certain types of changes and embrace others. ... Many methodology failures are caused by a problem definition followed by a solution design, with little analysis of whether or not the solution design fits the company or the project team's culture."

According to Moore (2000), there are four basic organizational cultures: cultivation, competence, collaboration, and control; to which he matches one of three methodology categories: rigorous (RM), agile (AM), and ad hoc or no methodology (NM). A cultivation culture is motivated by self-realization and can be illustrated by Silicon Valley start-up companies, to which fits the NM category. A competence culture is driven by the need for achievement; collaboration cultures are driven by a need for affiliation; and control cultures are motivated by the need for power and security. While the agile approach fits the competence and the collaboration cultures, the RM fits the control culture.

Highsmith (2002) adds another dimension and associates each methodology to a specific market phase. According to Highsmith, while the NM approach fits the initial phases of software product development, at later stages, when close interaction with customers is required, the AM approach fits better. During the Main Street market phase, the RM approach fits the best.

Task

Address connections between culture, development methodology, and market phase in global software development.

Case Study 12.1, presented in Chapter 12, Change, describes an organizational survey which aimed at revealing the organizational culture and development environment before the agile approach was introduced into the organization. As is explained in Chapter 12, Change, the purpose of such an organizational survey is twofold: first, to understand the current culture of the organization in general and the status of software development in the organization in particular; second, to discover whether the agile approach fits for the organization—if it is found that it might, the data gathered during the survey is used to determine how to initiate the introduction of agile software development into the organization. Such a survey can be used in global software development in order to learn about the different cultures involved and to use the survey's results to enhance communication and cooperation between the distributed teams.

10.5 Application of Agile Principles in Non-Software Projects

The agile approach fits not only for the development of software. Its main ideas can also be applied for the accomplishment of other, non-software tasks and projects. In Chapter 12, Change, for example, we will see how this idea has been applied to the transition process to agile development. Here, we illustrate this idea by describing how an agile process guided the writing of this book.

10.5.1 Case Study 10.2. Book Writing

Just as software projects are learning processes, so is book writing. Indeed, when we signed the contract for this book, we were familiar with the ideas of agile software development and with how to teach it; yet, in the book writing process, not only did we learn new ideas about agile software development, we also improved our understanding of how to teach it and how to present its main ideas in writing.

In what follows, we share how we used the agile approach for the process of book writing. Naturally, not all the agile practices can be applied in this process exactly as they are applied in software projects. The readers are invited to indicate the agile practices that are reflected by our activities.

At the beginning of the writing process, we set an initial table of contents and a short iteration schedule of two weeks, in each of which each of us wrote a chapter and gave it to the second co-author for review. At the end of each iteration, after we had reviewed the chapter written by the co-author, we met for feedback and further discussions about specific topics related to the book shape, orientation, and content.

After several chapters had been written, we stopped the actual writing, reviewed all the chapters we had completed, integrated them into one consistent format, and updated the future writing process. Sometimes we changed the main ideas to be presented in the chapters we were going to write next; sometimes we decided to change the chapter order; yet other times we decided to reorganize a chapter structure.

For example, according to our initial plan, the section that deals with learning environments in each chapter included only the description of the studio meeting of that week. In a later stage, we realized that this section should be broadened to provide readers a comprehensive understanding of the teaching and learning processes of agile software development. Another example: Chapter 7, Learning, included in the beginning also the topic of abstraction. As can be seen in the current book structure, abstraction has become the topic of Chapter 8, which discusses several agile practices from the perspective of abstraction.

In addition, we facilitated reflective sessions about the reasons that guided us to change our perspective on the book writing process and its structure and shape.

At the end of the writing process, we packed the book and delivered it to the publisher—our customer—to be distributed to you—our users. We hope that we accomplished our task successfully. We now plan the next release.

Tasks

1. This is only a short description of the book writing process. Other agile practices were used in this process. Can you suggest which ones would be suitable for writing a book?
2. Suggest other projects and tasks for which agile ideas and practices can be applied? What is common to all these projects and tasks?

10.6 Globalization in Learning Environments

The tenth studio meeting is dedicated to preparing the presentation of the second iteration product to be held at the next (eleventh) meeting (see Chapter 6, Quality, for more details about this last week of the iteration).

10.6.1 Teaching and Learning Principles

The following teaching and learning principles have already been presented in previous chapters. In the context of this chapter, their connection to globalization

is highlighted. In our list of teaching and learning principles, presented in Chapter 14, Delivery and Cyclicality, these principles are numbers 6 and 11.

10.6.1.1 Teaching and Learning Principle 6: Establish Diverse Teams

Diversity is easy to attain in a global software development environment. Still, while diversity is naturally achieved among teams, it is not always reflected within each team. See Chapter 9, Trust, for more details about diversity.

10.6.1.2 Teaching and Learning Principle 11: Emphasize the Software Development Approach in the Context of the World of Software Engineering

This principle directs us to emphasize connections to relevant trends in the world of software engineering. Since nowadays most organizations make use of distributed teams, teachers of software engineering should expose learners to such experiences and develop the learners' relevant capabilities.

Task

Suggest two additional teaching and learning principles from Table 14.4 presented in Chapter 14, Delivery and Cyclicality, and explain their application in distributed software development.

10.6.2 An Agile Perspective on the Book/Course Structure

Continuing the discussion presented in Chapter 7, Learning, about the contribution of the course structure to the learning process, we examine in this section how additional characteristics of this book, when used as a course textbook, are compatible with the agile approach. Specifically, in what follows, we examine how the book/course structure, as described in Chapter 1, Introduction to Agile Software Development, is especially suitable for teaching the Software Development Methods course within the framework of agile software development.

Iterative. As the development of agile software projects is based on iterations, the two course components—lectures and studio meetings—are iterative. Based on feedback and reflective processes, such a framework constitutes a learning environment that supports gradual understanding of the developed

product (in the case of agile projects) and of software development methods (in the case of the course). (See also Chapter 7, Learning.) This implies that the course topics are revisited several times during the course, in different ways, from different perspectives, and on different levels of abstraction. This is similar to the way agile software development supports customers and team members in their gradual understanding of the software requirements.

Multifaceted. The development of agile software projects increases awareness of the multifaceted nature of the process. Likewise, the course described in this book highlights the different facets of software development methods from the agile perspective. This has also been done by the HOT—Human, Organizational, and Technological—analysis framework.

Interactive. As the development of agile software projects is based on close interaction between the project's stakeholders, the course is based on close student–student, student–lecturer, and student–academic coach interactions.

Enhances reflective processes. An agile development process includes reflective sessions on different occasions, mainly at the end of the iteration and the end of the release. In a similar fashion, each chapter of this book includes reflective questions, and three chapters (Chapter 7, Learning, Chapter 11, Reflection and Chapter 14, Delivery and Cyclicality) are dedicated to reflective sessions.

These common features of agile software development and the course learning highlight the fact that in the case of a Software Engineering Methods course, the course structure and syllabus are shaped according to the perspective on software engineering that it addresses. In other words, as a course that highlights software development processes that are based on stages is usually organized according to the structure of such processes, a course that is organized around the agile approach, like the one described in this book, shares similar characteristics with the agile approach, such as the ones described above.

Task

What additional common characteristics do software projects and learning processes share in general? In particular, what additional common characteristics do agile software projects and the learning process of this particular book share?

10.6.3 Case Study 10.3. Follow-the-Sun with Agile Development

In general, distributed teams should deal with coordination and collaboration issues in order to work effectively (Jarvenpaa and Leidner 1998, Carmel and

Agarwal 2001). This section describes an experiment conducted in academia designed to investigate the "Follow-the-Sun" idea in an agile software development environment. Data related to this experiment is still gathered in parallel to the writing of the book.

According to Carmel[1] the follow-the-sun idea means: "Hand-off work from one site to the next as the world spins (USA to India, for example). This way you reduce the total time of development by 50% if you have two sites, and by 67% if you have three sites. Follow-the-sun is about speed!—cycle-time reduction, time-to-market reduction, duration reduction."

Carmel adds that a follow-the-sun project must satisfy the following five conditions:

1. at least two sites substantially separated by time zones;
2. high dependency between sites;
3. project set up with the objective of reducing duration;
4. successfully achieving duration reduction;
5. successfully achieving duration reduction using objective measures.

In what follows we describe the experiment of the application of the follow-the-sun idea in an agile environment.

Experiment goals: To investigate the Follow-the-Sun (FTS) idea in an agile environment. Specifically, to check the duration measure and calculate the percentages of time reduction, if any.

Experiment participants: Three groups of 7–8 students each participate in the experiment. One group is the FTS group which is divided into two teams of 3 and 5 students each. Two additional groups are the control (CO) groups: one group of 8 students is divided into two teams of 3 and 5 students each and one group of 7 students is divided into two teams of 3 and 4 students each. Two academic coaches, who have guided agile teams since 2002, supervise the development process: one coach guides the FTS group and one guides the two CO groups.

Experiment description: The three groups develop a software project on the same subject with the same functionality. The project subject is a simulator of processes and threads scheduler in the Unix operating system.

In the first three weeks all students learn the subject and the project functionality, prepare a high level design and a list of development tasks, understand their personal roles (see Chapter 2, Teamwork), and learn the integration environment.

[1] See Carmel's blog at http://errancarmel.blogspot.com/2007/09/follow-sun-call-for-more-research-and.html.

The project is developed in two iterations of five weeks each. In the first iteration each student in each of the three groups invests 50 development hours, and each team develops the same project scope. The two CO groups work without any constraint and the FTS group work with several time and communication constraints, as is outlined in what follows:

1. Simulate work in two non-overlapped time zones. The team of 3 students (team3) works from 21:30 to 11:00 while the team of 5 students (team5) works from 11:30 to 21:00.
2. The academic coach is in the same time zone as team3 for the duration of the first iteration. Weekly meetings in the first iteration are conducted with each team during its working hours, when meetings with team5 are carried out over the phone.
3. Team3 members should not directly contact or talk on the phone with team5 members, and vice versa.
4. All access to the integration server (CVS) and to the electronic forums should be performed during the working hours of each team, as is described in item 1 above.

The first iteration aims at checking that these rules are kept and that the CO groups work naturally in a framework of 12 hours as is predicted. Also, during the first iteration, the CVS repository and the electronic forums are checked to see what information about work patterns they can provide.

The second iteration aims at checking the duration measure. Given the same number of development hours for all groups and the same functionalities to be developed, the duration of the second iteration is set to be 5 weeks for the CO groups and 2.5 weeks for the FTS group.

10.7 Summary and Reflective Questions

1. Indicate at least three guidelines that can assist you in forming a software team of three distributed sub-teams for the development of a certain software product. Explain the rationale of each guideline. Will your guidelines change if you have to form two sub-teams? Four sub-teams? Why?
2. Indicate at least three agile practices that can support solving the main problems that exist in distributed software development environments. Explain the contribution of each practice.

3. Distributed environments are diverse in many ways, such as work attitude, value set, time sense, and communication style. What, in your opinion, are the advantages and disadvantages of diversity in such situations?
4. Analyze the concept of globalization within the HOT—Human, Organizational, and Technological—analysis framework.
5. Discuss the analogy between the following three processes: software development, learning this book, and our book writing process.

10.8 Summary

In this chapter we focus on globalization. We first explore how it is expressed in software development when teams work in a distributed manner on the development of a specific software product. A few agile practices are described as they are applied in such environments. We suggest that the agile approach fits global software development because of its high visibility, transparency, and tightness—characteristics that contribute to co-located software teams and therefore to distributed ones as well. Second, the application of agility beyond software projects, for the management of other processes, is also addressed in this chapter.

References

Allen TJ (1984) Managing the flow of technology: technology transfer and the dissemination of technological information within the R&D organization. MIT Press, Cambridge, MA

Carmel E (1999) Global software teams: collaborating across borders and time zones. Prentice Hall, Upper Saddle River, NJ

Carmel E, Agarwal R (2001) Tactical approaches for alleviating distance in global software development. IEEE Software pp 22–29

Cusick J, Prasad A (2006) A practical management and engineering approach to offshore collaboration. IEEE Software pp 20–29

Friedman TL (2005) The world is flat: a brief history of the twenty-first century. Farrar, Straus and Giroux

Herbsleb JD, Mockus A, Finholt TA, Grinter R (2001) An empirical study of global software development: distance and speed. In: Proceedings of the 23rd international conference on software engineering (ICSE). IEEE Computer Society Press, Los Alamitos, CA

Highsmith J (2002) Agile software developments ecosystems. Addison-Wesley, Reading, MA

Jarvenpaa SL, Leidner DE (1998) Communication and trust in global virtual teams. Organ Sci 10(6):791–815

Moore GA (2000) Living on the fault line: managing for shareholder value in the age of the internet. Harper Business, New York

Nirenberg J (2002) Global leadership. Capstone Wiley

Sahay S, Nicholson B, Krishna S (2003) Global IT outsourcing: software development across borders. Cambridge University Press, Cambridge

Sangwan R, Bass M, Mullick N, Paulish DJ, Kazmeier J (2007) Global software development handbook. AuerBach Publications, Taylor and Francis Group, New York

Sawyer S, Guinan PJ (1998) Software development: processes and performance. IBM Syst J 37(4) http://www.research.ibm.com/journal/sj/374/sawyer.html

Yourdon E (1997) Death march: the complete software developer's guide to surviving "mission impossible" projects. Prentice Hall PTR, NJ

11
Reflection

Abstract

This chapter describes the notions of reflection and retrospective: reflection usually refers to the individual's thinking about what he or she has accomplished; retrospective is usually conducted in teams, and is partially based on the individuals' reflections performed during the retrospective sessions. In fact, you, the readers, are familiar with and have experienced these notions at the end of the first iteration of the book, in Chapter 7, Learning, both individually and on the team level. Indeed, since these concepts are not trivial to grasp, it is preferred that learners experience them first, before the theoretical ideas are presented. This chapter, which closes the second iteration of the book, serves as an opportunity to understand the theory behind these concepts as well as to add some practical details about their actual performance.

11.1 Overview

This chapter focuses on the nature of reflective processes on the individual level (reflection) and on the team level (retrospective). While reflection provides the individuals feedback with respect to how they perceive different aspects of the development process, retrospective elevates these thoughts to the team level.

Reflective thinking is important in learning processes in general and software development processes in particular (Schön 1987, Hazzan 2002). Specifically,

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_11, © Springer-Verlag London Limited 2008

since software development can be viewed as a learning process (see Chapter 7, Learning), agile teams and the other project stakeholders facilitate reflective sessions on a regular basis in order to improve their understanding of the developed product and process. In this spirit, we have met the concept of reflective thinking in this book several times so far. For example, at the Business Day, conducted at the end of each iteration (see Chapter 3, Customers and Users), a specific time slot is allocated for a reflective session; at the end of the first iteration of the book (Chapter 7, Learning), the reflective thinking focused on learning processes.

In order to enable readers to start implementing retrospective sessions by themselves, we present in this chapter, in addition to the general descriptions of the two reflective processes—reflection and retrospective—specific guidelines with respect to the facilitation of retrospective sessions.

11.2 Objectives

- Readers will understand the importance of reflective processes in software development environments.
- Readers will become familiar with the notions of reflection and retrospective and how to employ their benefits.
- Readers will learn basic ideas with respect to the facilitation of the retrospective process.
- Readers will increase their awareness with respect to the variety of opportunities that agile software development offers for reflective processes.

11.3 Study Questions

1. What were the main lessons you learned from the reflective sessions you facilitated in Chapter 7, Learning? How did you use these lessons in the continuation of the book learning? How did you use them in the continuation of your current software development project?
2. In your opinion, with respect to which topics presented so far in the book should you improve your understanding? What characterizes these topics? How will you improve your understanding about these topics?

11.4 Case Study 11.1. Reflection on Learning in Agile Software Development

This case study illustrates how reflective processes can be facilitated from the early stages of software projects. Data are taken from a team undergoing transition to agile software development which carries out a one-hour reflective session at the end of each iteration. Additional details about this team are described in Case Study 4.4 (Chapter 4, Time). See also Hazzan and Dubinsky (2007).

The first reflective session took place at the end of the first two-week iteration, as part of the iteration summary meeting (see Chapter 3, Customers and Users), and it focused on learning. The software teammates, the customer, and management representatives were asked to reflect on what they had learned during the first iteration with respect to the following topics: the software development process, the project, the teammates, roles in the team, work habits, and any additional topics they saw fit to raise. They were asked to elaborate on what they learned and to specify how they learned it; in other words, what in the agile software development environment enabled their leaning.

Specifically, by filling in Table 11.1, each participant first addressed the following question: "What did you learn during the first iteration?"—which calls for reflective thinking.

To strengthen the message delivered in Chapter 4, Time, we present here responses related to time management expressed with respect to the *software development processes* and the *work habits* categories.

Software development processes:

- It is important to estimate times for each task and then to specify how long it took in practice.
- It is difficult to estimate times precisely.

Table 11.1 Reflection on learning (© [2007] IEEE)

About	What did you learn?	How did you learn it? What in the development environment enabled that learning?
Software development processes		
The project		
The teammates		
Roles in the team		
Work habits		
Additional topics—please elaborate		

Work habits:

- We can follow a schedule.
- Changes are more focused and there was less waste of time.
- Time management enables us to produce the maximum benefit from each person.
- I saw how the time management process was internalized by all team members.
- When we focus on a task with targets and when we limit the times, we achieve better results.
- Learning is a nice process. The question is for how long, and how can its output be measured.

In order to establish a learning community, each practitioner shared with the other participants one item he or she had learned during this personal reflection. To deliver the message about learning, we present one illustrative quote: "I have the feeling that within a short time we can see what we have done. I can see the direction. Even if it is not accurate, it is a direction."

Tasks

1. Conduct the above reflection with respect to the project you are currently working on.
2. Ask your team members to carry out Task 1. Compare your reflections. What can you learn from this comparison?
3. What does the last quote reflect with respect to the nature of the learning process? Can you connect your answer to the constructivist perspective described in Chapter 7, Learning?
4. In your opinion, how did the fact that the team facilitated this reflection in an early stage of the development process influence the continuation of the development process?

11.5 Reflective Practitioner Perspective

Reflection is the process by which an individual examines his or her actions during the accomplishment of a task or after the task has been accomplished. Though reflection is not a new concept, its common practice has been boosted after Schön

published his two books, *The Reflective Practitioner* in 1983, and *Educating the Reflective Practitioner* in 1987 (Schön 1983, 1987). In order to become a reflective practitioner, one should keep reflecting on his or her accomplishments, activities, and behaviors. Schön's books advocate the idea that a person who keeps reflecting becomes a reflective practitioner, a position which enables him or her to keep improving his or her professional skills.

HOT

Generally speaking, the reflective practitioner perspective encourages professional practitioners (such as architects, managers, musicians, and others) to examine and rethink their professional creations during and after the act of creation. The working assumption is that such a reflection improves both proficiency and performance within such professions. Analysis of the field of software engineering and the kind of work that software engineers usually accomplish in general (Schön and Bennett 1996), and the agile approach in particular, support the adoption of the reflective practitioner perspective in software engineering processes (Hazzan 2002, Hazzan and Tomayko 2003). Specifically, a reflective mode of thinking may improve the performance of some of the agile practices.

Though the importance of reflective processes is acknowledged by many professions, it is not always done if time is not specifically dedicated for this process. Accordingly, since agile software development acknowledges the importance of reflective processes, it allocates specific time slots for their accomplishment. One time slot is a retrospective that usually takes place at the end of the release (see the remainder of this chapter and Chapter 14, Delivery and Cyclicality).

Tasks

1. What topics, in your opinion, are suitable as subjects for reflective thinking?
2. For each of the following topics, suggest a specific subject on which, in your opinion, it is worth facilitating a reflective process:
   - the developed software systems;
   - the development environment;
   - the development process;
   - the ways algorithms are used;
   - ways of thinking;
   - quality issues.
3. What—Human, Organizational, and Technological—aspects might elicit reflective thinking? Illustrate your answer.

Though it takes time to become a reflective practitioner and it requires some practice and experience, this skill can be learned and gained gradually. One possible way to start practicing being a reflective practitioner is by introducing it to your team in a team setting, for example, in retrospective sessions.

11.6 Retrospective

Retrospective is a reflective session that takes place on the team level, usually during long sessions (from one hour to several days). In retrospectives, in addition to personal reflective processes, the team as a whole facilitates reflective thinking to derive lessons from its past experience. As mentioned earlier, in Chapter 7, Learning, you have already experienced retrospective processes.

Though the concept of retrospective usually refers to long sessions that take place at the end of the release, we adopt this notion for any *team* gathering (such as the end of the iteration meetings, role-holders meetings, and more) whose aim is to reflect on the team performance in order to improve the software development process and product. We note that the term "team" refers not only to the development team; rather, it encompasses also, if needed, the customer, management, and other project stakeholders.

Several systematic approaches have been suggested on how to conduct effective retrospective processes in agile teams. The most widely known are post-iteration workshops (Cockburn 2001, Salo et al. 2004) and the postmortem review technique (Dingsøyr and Hanssen 2003, Myllyaho et al. 2004). Empirical studies on the effectiveness of these methods have been conducted by Salo (2004, 2005), Salo et al. (2004), and a few experience reports have been published as well (Lamoreux 2005, Talby et al. 2006).

In the continuation of this section we present main ideas and guidelines relevant to the facilitation of and participation in retrospective sessions.

Retrospective processes guide a team reflection, in which each team member shares his or her reflection with the other participants in order to improve the team performance, and consequently, the process and product quality. Accordingly, communication and feedback are important in retrospective sessions, and dependencies and mutual feedback should be enhanced in such sessions.

Acknowledging the importance of the learning process, agile methods dedicate specific time slots for retrospective sessions: at the end of each short iteration a short retrospective session takes place during the Business Day (see Chapter 3, Customers and users); at the end of the release a longer retrospective session is facilitated (see the continuation of this chapter and Chapter 14, Delivery and Cyclicality).

Since during the retrospective session development/planning/design/testing and other kinds of tasks are not accomplished, the mere existence of retrospective

sessions delivers a very important message about their importance. This message is based on the anticipated contribution of retrospective sessions to the team's future performance and to the product and process quality. In other words, it is assumed that the time invested in retrospective sessions will be repaid in improved product and process quality.

Further, the experience the participants gain in such retrospectives is cumulative. In other words, each time a team is engaged in a retrospective, new lessons are learned, which on the one hand may deepen previous lessons, and on the other hand may open the team learning to new horizons.

Tasks

1. In Chapter 3, Customers and Users, the Business Day and its components are described. Explain the importance and contribution of the reflective session to the entire Business Day. How does each of the elements of the Business Day which take part before the reflective session set the atmosphere for the reflective session?
2. It is sometimes argued that a retrospective session is a waste of time. Can you explain the source of such claims? What is your opinion with respect to such statements? How would you cope with expressions that claim that retrospective sessions waste time and that it is better to dedicate that time for development tasks?
3. Suggest an example of an idea that may emerge during a retrospective session and that has the potential to improve team performance in the future.
4. If you have already participated in retrospective sessions, describe their dynamics. Reflect on how the retrospective session influenced the continuation of the development process.
5. What feelings might practitioners have in a retrospective meeting? What are their resources? How can they influence the atmosphere in the retrospective session?

11.6.1 The Retrospective Facilitator

Each retrospective session should be facilitated by a moderator. One option is to invite a facilitator who is not part of the team. Another option is to assign one of the team members to be the retrospective facilitator. Teams which facilitate retrospective sessions on a regular basis can either add the role of Retrospective Facilitator to the role scheme presented in Chapter 2, Teamwork, or add this

responsibility to one of the other roles on the team (for example, to one of the role holders in the leading group). Alternatively, the Retrospective Facilitator role can be rotated among the team members. It is recommended, of course, that the Retrospective Facilitator know how to facilitate retrospective processes. However, even if the team does not have a person who is familiar with guiding retrospective processes, the team can dedicate the needed time for a gradual improvement of its retrospective sessions in a constructivist manner.

The role of the Retrospective Facilitator includes the selection of a subject for the retrospective, in coordination with the team and the team leader, and the actual facilitation (including time keeping) of the retrospective meeting itself. During the retrospective, special attention should be given by the facilitator to the fact that all the participants are active and highly communicative.

Tasks

1. Describe the benefits and disadvantages of assigning one of the team members to be the Retrospective Facilitator.
2. Describe the benefits and disadvantages of having a Retrospective Facilitator who is not one of the team members.
3. In your opinion, are there situations in which it is better to have a Retrospective Facilitator who is not part of the team, and other situations in which it is better that this role is carried out by one of the team members? Explain and elaborate on these different situations.
4. How can the retrospective practice empower agile teams?

11.6.2 Case Study 11.2. Guidelines for a Retrospective Session

The following list presents guidelines (adopted from Talby et al. 2006) formulated by a specific team for its retrospective sessions that take place at each of its Business Days.

- Only one specific problem is discussed at each retrospective meeting.
- The problem discussed should relate to the development process, not the developed product.
- The subject is chosen in advance by the moderator (after informal/formal consultation with other team members), and presented at the beginning of the retrospective meeting.

- The retrospective cannot exceed one hour.
- The whole team is required to attend the retrospective.
- Everyone is proactively encouraged to speak, but is not required to do so.
- Team members are encouraged to speak their own opinions.
- The moderator records important insights and proposes action items that surface during the meeting.
- The moderator summarizes the meeting by reading to the team the action items that have been decided upon.
- The moderator publishes the main insights and action items for the team soon after the retrospective. A wiki can be used for this purpose.
- The decided action items are effective immediately. These are changes in the day-to-day team operations that should help resolve the debated problem.

Tasks

1. Express your opinion with respect to each guideline presented above. Explain the rationale behind each guideline as you conceive of it. In your opinion, what advantage and/or disadvantage does each guideline have? In what way may each guideline help achieve the retrospective goals? In what way (if at all) does each element foster agile culture?
2. Formulate with your team the guidelines with which the team wishes to facilitate its retrospective sessions.
3. According to the guidelines presented above, how does the team conceive the role of the Retrospective Moderator?

11.6.3 Application of Agile Practices in Retrospective Sessions

This section highlights several facilitation guidelines for retrospective sessions in agile software development environments. These guidelines demonstrate that when a retrospective session takes place in an agile project, the retrospective itself should be based on agile ideas—for example, it should foster diversity (see Chapter 9, Trust), support learning processes (see Chapter 7, Learning), and include the whole team. In other words, when a retrospective takes place in an agile software development environment, it should apply and promote agile

practices and principles. The idea that agile methods apply beyond the development of software products has already been discussed in Chapter 10, Globalization.

Whole team. Everyone who belongs to the team should participate in the retrospective. Also, it is recommended that the team take an active part in the preparation of the retrospective as well as during and after it, when lessons are implemented.

Abstraction. During the retrospective meeting, it is recommended that the topics representing different levels of abstraction (see Chapter 8, Abstraction)—from conceptual ideas to practical activities and measures—be addressed. Further, it is recommended that this movement between abstraction levels be emphasized, to enable the participants to exploit the associated cognitive benefits.

Diversity. Everyone should be encouraged to share his or her professional thoughts (see also Chapter 9, Trust).

HOT

Measures. It is important to accompany the application of each decision made in a retrospective session by a measure that first, will enable the team to observe whether or not the decision itself is applicable; and second, will enable the team to examine its actual performance and contribution to the development process. See additional details about measures in Chapter 5, Measures.

Time allocation. As with other activities, time should be allocated for the retrospective session as well. Kerth (2001:53) suggests two or three days for a retrospective session. Since we expand the use of the term retrospective to include each reflective activity which is conducted at the team level, the time for a retrospective should be allocated according to the retrospective scope and goals. For additional discussion about time related issues in agile software development environments, see Chapter 4, Time.

Tasks

1. What additional agile characteristics can and should be expressed in retrospective processes?
2. How does each of these characteristics enhance the retrospective process, if at all? Illustrate your thoughts with examples.
3. Describe a problem that in your opinion is an appropriate subject for a retrospective session. Facilitate a retrospective session with your team about that problem. Make sure that a measure that indicates whether the solution is applicable or not is set up to determine the solution's effectiveness.

11.6.4 End of the Release Retrospective

This section suggests a framework for the end of release retrospective. The framework should be adjusted for each specific team's and project's needs. The place and role of this retrospective at the end of the release period is described in Chapter 14, Delivery and Cyclicality. For additional details about the facilitation of retrospective sessions with software teams, look at Kerth's book *Project Retrospective* (Kerth 2001).

Retrospective Place. It is advisable to facilitate the release retrospective outside the development site. The idea behind this recommendation is first, to disconnect the practitioners from their ongoing work in order to enhance reflective thinking, and second, to demonstrate that the retrospective is at least as important as the development work. This importance is highlighted by allocating a special time and place for the retrospective session, as is done for other kinds of tasks.

Retrospective Length. In Chapter 3, Customers and Users, we saw how a retrospective session is integrated on a weekly (or bi-weekly) basis into the iteration summary meeting that takes place on the Business Day. For the end of release retrospective a longer period of time should be dedicated.

The retrospective scope is determined by its length. In a short retrospective session that takes place on the Business Day, one topic at most can be addressed; for a retrospective that takes place at the end of the release, in which the team wishes to get a comprehensive understanding of the release and in order to collect the different elements and lessons learned during the release into one framework, a longer period of time should be allocated.

In addition, the longer period of time which is allocated for a release retrospective gives the team a timeout before the next release starts. Therefore, two days seems to be an appropriate period.

Retrospective Participants. The retrospective participants should be selected according to the retrospective's goal. The entire retrospective (as well as parts of it) may include only the development team, or the team with the management and/or customer. The exact mix should be determined according to the team climate and dynamics, the development stage, and the lessons learned in previous retrospective sessions. In any case, when a specific decision is made about participation, its rationale should be shared with all the project stakeholders.

Topic(s) Selection. In order to address most of the team members' concerns in the retrospective, it is recommended that the retrospective subject(s) be selected from a list that is generated by the team members, posted on the walls of the informative workplace, and accessible to all.

This topic selection process has several advantages. First, the subject will be relevant for at least several team members; second, it is reasonable to assume that a topic selected in this way will be connected to the daily project life; third, time will not be spent in the retrospective meeting deciding on the subject of the retrospective; fourth, such a selection process will enhance the environment's transparency. Yet in some cases, the team leader or a project manager may suggest topics which were not selected democratically.

From the suggested list of topics, it is advisable to select topics about which different opinions have been expressed and to avoid selecting a topic that involves personal quarrels and accusations.

Retrospective Preparation. The participants should be encouraged to bring to the retrospective session ideas, event descriptions, measures, and personal stories related to the retrospective scope. To encourage the participants to start preparing themselves for the retrospective, they can be encouraged to think about one positive experience they have had during the release and one experience they have bad feelings about.

The Retrospective Facilitator should be aware of the different concerns that the practitioners may bring into the retrospective. Global planning should be constructed accordingly; yet some freedom should be left to enable the accommodation of the retrospective timetable to the participants' needs, as well as unexpected events that may come up during the retrospective.

Retrospective Organization. As in agile software development processes, it is advisable to base the retrospective session on cycles, each of them including a trigger (explained below), a group activity, a discussion, and a summary.

If the retrospective participants break into subgroups for different activities, the subgroups' members should be mixed for each activity, in order to allow all the retrospective participants to interact with as many other participants as possible. The gathering of all the participants after the group activities, in which the groups report their conclusions to the entire retrospective milieu and a discussion is facilitated, is important and should not be skipped.

As with the development process, in which development continues according to the current understanding, both from the customer and the team perspective, of what should be developed, so the contents of the next retrospective cycle can be determined during the retrospective according to general guidelines prepared in advance. Nevertheless, the Retrospective Facilitator should navigate the retrospective according to his or her experience and conception of the current situation and atmosphere.

Retrospective Trigger. A trigger is a stimulus that fosters thinking on selected topics. A well chosen trigger can open the participants' horizons to new ideas and enable them to communicate those ideas from new perspectives. It is

advisable, though not necessary, to employ the trigger in small groups, since that will enable each participant to express his or her opinion in a more relaxed atmosphere. It will also enable the retrospective participants to be exposed to a wider perspective on the topic, since after the trigger is employed in small groups, the subgroups present their conclusions to the full forum.

There are different kinds of triggers. Since they vary in the time it takes to facilitate them, the Retrospective Facilitator should select them according to the goals of the retrospective and the available time.

Among many options, movies can serve as triggers. For example, a movie about a leader or about a natural phenomenon can serve as triggers that stimulate interesting discussion. After the movie is shown, the similarities and differences between what is seen in the movie and what happens in software engineering processes can be discussed. In other words, the question that can be addressed is: To what extent is the movie a good metaphor (see Chapter 3, Customers and Users) for software development processes?

Movies are good triggers for retrospective sessions because they enable each team member to connect what he or she watches to his or her professional life experience. In addition, such a trigger encourages diversity, since each teammate brings into the discussion his or her particular perspective, experience, and background.

Task

Review different movies you have watched in your life. What lessons, if any, did they attempt to deliver? Can these lessons be transferred and used in software development processes?

Example for a Trigger: *March of the Penguins.* We illustrate how to use movies as triggers for retrospectives about software development processes by the National Geographic feature film *March of the Penguins.* The main reason for the selection of this movie is that it emphasizes several ideas about the nature of software development. Specifically, since software development is characterized by changes (See Chapter 12, Change), the movie illustrates how nature adjusts itself to changes.

The movie describes the yearly journeys of the penguins of Antarctica to their ancestral breeding grounds. There, the penguins participate in a courtship that, if successful, results in the hatching of a chick. For the chick to survive, both parents must make multiple arduous journeys between the ocean and the breeding grounds.

Tasks

1. Explore different resources about this movie. What messages are delivered when the movie is described?
2. Watch the movie. What relevant lessons for software development can be learned from the movie?
3. In what follows, several facts taken from the movie are listed. For each of them explain how it is or is not related to software development in general and to agile software development in particular.
   - Each time one penguin leads the march.
   - The goal (direction) is clear (to find a mate); the path changes, since the land moves beneath the penguins' legs each year. For example, when ice starts melting it has more holes. Since the food is available beneath the ice, such moves enable the penguins to find food.
   - When the temperature goes down, the penguins adjust their behavior to the change.
   - The term "unified and cooperative team" is used in the movie.
   - The penguins keep shifting places so each time another penguin is in the middle of the group and keeps its body warm.
   - Penguins have an internal compass.
   - The penguins look for safe ground.
   - In the winter the penguins are not able to survive alone; they stand next to one another to conserve heat.
   - Pair parenting: the mother transfers the egg to the father when she goes to eat; the father keeps the egg for two months; both the father and the mother feed the chick.
   - Taking care of the egg: one parent keeps the egg and the second one supports it. They rotate roles: when one goes to eat, the second keeps the egg/chick.
   - When an egg is broken, the mates separate.
   - Problem solving: There is food beneath the ice that cannot be accessed; what can be done?
   - There is one new egg each year for each pair.

- Not all the chicks survive their first year of life.
- For four years the chicks do not participate in the march, they start it only in their fifth year.

4. How can the lessons you learned from *The March of the Penguins* help your team improve its performance?
5. Think of another movie, not necessarily about animals, that can illustrate agile ideas. Elaborate these ideas. In what way are they expressed in the movie?

11.7 Reflection in Learning Environments

This studio meeting summarizes the second course iteration and opens the third iteration. The main objectives of this meeting are to present to the customer what has been developed during the second iteration, to provide and receive feedback, and to begin the third iteration by listening to the customer's priorities with respect to what he or she wants developed in the third iteration. Also in this meeting, both personal reflection and team retrospective take place.

11.8 Summary and Reflective Questions

1. Reflect on your presentation of the second iteration product to the customer. What went well? What can be improved?
2. What main lessons did you learn from the development of the second iteration?
3. So far, you, the reader, have been trained to perform reflective processes by being asked to reflect on the software development process on a weekly basis. Together with your team, collect the different lessons you learned from these weekly reflections. Discuss how you can use these lessons in the continuation of the development of your current software project as well as for the development of future software projects.
4. Together with your team, select a topic for a retrospective session of one hour. Choose one team member to facilitate this session. Choose another topic for a retrospective session of one hour and another team member to facilitate it.

Observe these two retrospective sessions. Discuss the facilitation style of the two team members. What can they learn from each other?

5. How are reflective processes—reflection and retrospective—connected to the Agile Manifesto presented in Chapter 1, Introduction to Agile Software Development?
6. How can the retrospective outcomes be measured?
7. In your opinion, should an iteration/release retrospective take place both when the release ends successfully and when it fails? Explain your opinion. What advantages and disadvantages would a release retrospective have in each case?
8. A reflective practitioner reflects at almost any opportunity. Review the different agile practices and ideas presented so far in this book and discuss their fitness and their potential to promote reflective processes—either reflections or retrospectives.
9. Analyze the reflection and retrospective activities within the—Human, Organizational, and Technological—analysis framework.

11.9 Summary

This chapter looks at the contribution of reflective processes—reflection and retrospective—to software development. One of the main messages delivered in this chapter is that these practices should be conceived important as development tasks are and treated like other kinds of tasks in agile software development environments: that is, with specific time allocations, the inspiration of the agile approach, and the application of agile practices.

References

Cockburn A (2001) Agile software development. Addison-Wesley, Reading, MA

Dingsøyr T, Hanssen GK (2003) Extending agile methods: postmortem reviews as extended feedback. In: 4th international workshop on advances in learning software organizations LNCS 2640, Springer, New York, pp 4–12

Hazzan O (2002) The reflective practitioner perspective in software engineering education. J Syst Software 63 (3): 161–171

Hazzan O, Tomayko J (2003) The reflective practitioner perspective in eXtreme programming. Proceedings of XP agile universe, New Orleans, Louisiana, USA, pp 51–61

Hazzan O, Dubinsky Y (2007) The software engineering timeline: a time management perspective. Proceedings of the IEEE international conference on software—science, technology & engineering, Herzelia, Israel, pp 95–103

Kerth NL (2001) Project retrospectives: a handbook for team reviews. Dorset House Publishing Company, New York

Lamoreux M (2005) Improving agile team learning by improving team reflections. Proceedings of Agile, Colorado

Myllyaho M, Salo O, Kääriäinen J, Hyysalo J, Koskela J (2004) Analysis of small and large post-mortem review methods. Proceedings of ICSSEA: 17th international conference on software & systems engineering and their applications, Paris, France

Salo O (2004) Improving software process in agile software development projects: results from two XP case studies. In: EUROMICRO 2004. IEEE Computer Society Press, Rennes, France

Salo O, Kolehmainen K, Kyllönen P, Löthman J, Salmijärvi S, Abrahamsson P (2004) Self-adaptability of agile software processes: a case study on post-iteration workshops. Proceedings of XP, Germany, pp 184–193

Salo O (2005) Systematical validation of learning in agile software development environment. 7th international workshop on learning software organizations, Germany

Schön DA (1983) The reflective practitioner. BasicBooks

Schön DA (1987) Educating the reflective practitioner: towards a new design for teaching and learning in the profession. Jossey-Bass, San Francisco

Schön D, Bennett J (1996) Reflective conversation with the materials. In: Winograd T, Bennett J, De Young L, Hartfield B (eds) Bringing design into software. ACM Press, Addison-Wesley Publishing Company, Boston, pp 171–184

Talby D, Hazzan O, Dubinsky Y, Keren A (2006) Reflections on reflection in agile software development. Proceedings of the agile conference, Minneapolis, Minnesota, USA, pp 100–110

12
Change

Abstract

Coping with change is one of the main challenges of software engineering. This challenge encourages agile software developers to establish a development process that enables them to cope successfully with changes introduced during that process, while keeping the high quality of the product. This chapter focuses on change introduction into organizations that plan to transit, or that are already in the transition, to agile software development. Specifically, we present an evolutionary framework for coping with change, using it for understanding a transition to agile development. We also suggest several methods to use in such a transition process, such as an organizational survey and a condensed workshop format on agile software development.

12.1 Overview

The notion of change has been mentioned so far in the book several times. For example, in Chapter 1, Introduction to Agile Software Development, we reviewed the Agile Manifesto in general and one of its main ideas—responding to change by following a plan—in particular. We emphasized that this idea encourages agile software developers to establish a development process that enables them to cope successfully with changes introduced during that process, while keeping the high quality of the product. Later, in Chapter 3, Customers and Users, and in Chapter 4, Time, we learned how, in practice, agile software development

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_12, © Springer-Verlag London Limited 2008

methods inspire a process that enables change introduction in the developed product which emerges from an improved understanding of the software requirements. This characteristic of agile development is also reflected by the agile approach's acknowledgement of learning processes (see Chapter 7, Learning). Further, in Chapter 8, Abstraction, we saw how the agile approach legitimizes change introduction in the code design, by allowing emerged design and ongoing refactoring activities to be revealed based on an improved understanding by the team members of the code structure and functionality.

In this chapter we deepen the examination of the concept of change, highlighting the fact that the agile approach supports changes of different kinds. This legitimization is important in software engineering processes, first, because of the frequent changes in the discipline's body of knowledge itself and in its application; and second, since changes are inherent elements of software development processes and therefore should not be neglected.

We focus on two kinds of changes. The first one—changes in the customer's requirements—has already been addressed in the book (see Chapter 3, Customers and Users). The second kind of change is the organizational required when the agile approach is introduced. For each kind of change (software requirements and organizational change) we first present the importance of supporting the given change. Then, in order to explain how the agile approach copes with that change, we use Plotkin's framework, borrowed from evolutionary theories, which describes how the universe copes with changes throughout its evolution. This exploration shows that agile software development realizes that changes are an inherent part of any software development process, and therefore it adopts several ways that support change instead of blocking it.

12.2 Objectives

- Readers will understand the concept of change and its centrality in the software development process.
- Readers will review different kinds of changes which are inherent in software development.
- Readers will understand the need for and nature of the change required when agile software development is introduced into the organization.
- Readers will become familiar with several ways to initiate and maintain the agile software development process.

- Reader will gain practical advice with respect to organizational surveys and short presentations of agile software development.

12.3 Study Questions

1. What adjectives, in your opinion, fit for a description of the agile approach (besides agile, of course)? Why? What do these adjectives reflect?
2. What kinds of changes are inherent in software development processes? Explain their source.
3. According to what you have learned so far in the book, how do agile teams cope with changes introduced by the customer?
4. Predict what claims are typically raised by practitioners who do not wish to transition to agile software development. Do these claims contradict the Agile Manifesto? How would you cope with such expressions of resistance?
5. How could you learn about the software development process in a software organization? Suggest different methods that you would employ.
6. If you had to advise a software organization which does not work according to the agile approach how to transit to agile software development, what would be your three most important pieces of advice? Explain their rationale.

12.4 A Conceptual Framework for Change Introduction

Henry Plotkin presents the notion of change in his book *Darwin Machines and the Nature of Knowledge* as part of the chapter that deals with the evolution of intelligence.

> Change is a universal condition of the world. If the world were unchanging, then evolution would have proceeded to some optimal point and then ceased. This has not happened. Nothing stands still, and the very occurrence of evolution is both a force for change itself and proof positive for its existence (Plotkin 1997, 139).

Since software development processes, as has been explained earlier in this book, are also characterized by changes, it might be a good idea to use some evolutionary mechanisms for coping with changes for the shaping and analysis of software development processes. In the rest of this chapter we apply this idea by using Henry Plotkin's notion of coping with change.

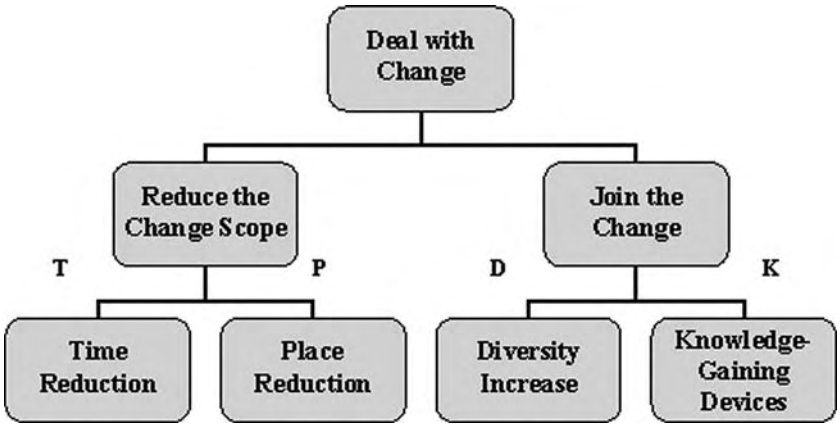


Figure 12.1 *Solutions for dealing with change (Plotkin 1997) (© [2003] IEEE).*

The main question Plotkin poses is how we can deal with the uncertainty introduced by change. He describes two main sets of solutions to deal with change and explains how they enable us to cope with change (Plotkin 1997, 145–152).

The first set of solutions concerns with "reducing the amount of significant change," thus reducing the change scope (see left branch in Figure 12.1). One way to do this is by *reducing the period of time* (see branch T in Figure 12.1) between conception and reproductive ability. It means to keep the ratio of life-span length to numbers of offspring low, or in other words, to maintain high reproductive output in a relatively short period of time. This copes with change by keeping the genetic instructions for each individual updated as much as possible.

Task

Which agile practices lead to time reduction?

The second way to reduce the amount of significant change according to Plotkin is to *live in a relatively isolated and unpopulated place* (see branch P in Figure 12.1). A variation on this idea is parents protecting their offspring by isolating them.

Task

Which agile practices lead to place reduction?

The second set of solutions for the phenomenon of change takes the form of "if you can't beat it, join it," i.e., change the phenotypes so that they can change with

and match the changing features of the world (see right branch in Figure 12.1). The first strategy to accomplish this target is to *produce large numbers of different offspring* in order to increase *diversity*. Thus, the chance increases that this approach will lead to a situation in which at least some individuals will be able to face the change (see D branch in Figure 12.1).

Task

Which agile practices enhance diversity?

The second strategy, called the "tracking option," enhances change within phenotypes by *producing phenotypes that change in response to changes in the world* (see branch K in Figure 12.1). The tracking option is supported by knowledge-gaining devices which, according to Plotkin, are the immune system and the brain intelligence mechanisms. The immune system operates in the sphere of *chemistry,* while the brain mechanisms, known as rationality or intelligence, operate in the sphere of the *physical world* of temporal and spatial relationships of events and objects.

Task

Which agile practices can be considered as knowledge-gaining devices?

We now apply Plotkin's framework for the analysis of two kinds of change which are relevant for software development environments: a change in software requirements, and organizational change. We first re-examine how agile software development copes with changes in the software requirements. At this stage of the book, readers will be familiar with most of the ideas presented in this examination. Then we apply Plotkin's framework for organizational changes required when an organization moves to agile software development.

12.4.1 Changes in Software Requirements

Changes in software requirements are predominant partially due to the fact that software is an intangible product. On the one hand, because it is an intangible product, it is difficult to envision how the software will look, work, function, and evolve; accordingly, customers keep changing their requirements as they improve their understanding of the features of the product they need and ask for. On the other hand, software intangibility suggests incorrectly that change introduction in software products is easy, simply because it is applied on an intangible product. This, of course, turns out to be mistaken; change introduction in software products indeed causes complications, predicted as well as unpredicted.

Since, however, change introduction in software products is an integral part of software development, we have two options. One option is simply to neglect this fact and to block change introduction in the requirements after they have been formalized, set, and agreed upon; the results of this approach are well known. The second option is to offer a development process that allows changes in the requirements without reducing the software quality. This is what agile software development attempts to accomplish.

In what follows we summarize several of the ideas we have already examined in the book with respect to software requirements, presenting them within Plotkin's framework.

12.4.1.1 Reducing the Change Scope—Time Reduction Mechanism

The customer has an opportunity to update the requirements at the end of each *short release and iteration;* this clearly is a mechanism for time reduction. The question to be asked is how this mechanism influences cost and quality.

Analysis shows that the cost of change introduction in this fashion remains constant. This is because it allows updating the requirements on a small scale as soon as it is realized that a different feature is needed. Under this working assumption, customers do not assume what will probably be needed, and therefore, in the planning sessions, ask only for relevant requirements.

This approach stands in contrast to situations in which customers are told that "the later a change is introduced, the more expensive it will be." In such situations, customers (and teammates) try to make (and implement) a "complete" list of requirements, even though it may turn out at a later stage, when their understanding of the project requirements is clarified, that not all these requirements are indeed needed. However, if change introduction is not allowed, in order to save money and to feel on safer ground, customers will attempt to provide a priori a full set of requirements, without a solid basis for their decisions.

Task

How can this last described behavior be explained by the ideas presented in Chapter 9, Trust?

Since in the agile development process the cost of change introduction is constant during the entire process, customers are not forced to present a full requirement list without a clear idea of their needs. Thus, at the end of an agile project, only features that the customer needs will have been developed.

Indeed, there are cases in which customers keep changing their requirements on a regular basis, and it becomes difficult to cope with such frequent changes. Such cases further emphasize the need for customer collaboration with the relevant role holders whose technical recommendations should be heard, shared with the customer, and considered. In any case, the customer makes the final decision.

Task

Which other agile practices leave the cost constant and the quality high? Can they be considered as time-reduction mechanisms?

12.4.1.2 Reducing the Change Scope—Space Reduction Mechanism

Space is reduced by the co-location of the team and the customer in the workspace. This space includes also the walls, which serve as a communication means, so that all the relevant information is accessible to all. In the case of distributed teams (see Chapter 10, Globalization), a *virtual space reduction* should be supported—for example, by using video conference calls and/or a shared project wiki.

Tasks

1. How does space reduction enable developers to cope with changes in software requirements?
2. Explain the need in space and time reduction within the HOT—Human, Organizational, and Technological—analysis framework.

Space and time reduction in agile software development create a situation in which changes take place within a defined framework, which increases the participants' confidence in the change process they go through.

12.4.1.3 Join the Change—Diversity Mechanism

Naturally, the discussion about diversity, presented in Chapter 9, Trust, is of high relevance in this context of change in software requirements. This is because diversity welcomes new ideas and perspectives that are so predominant in change introduction processes.

Task

Review the concept of diversity presented in Chapter 9, Trust. How does diversity welcome change introduction? How can it interfere?

12.4.1.4 Join the Change—Mechanism of Knowledge-Gaining Devices

Several agile practices that can be characterized as knowledge-gaining devices have been presented so far in this book. Among them, we have mentioned short releases and iterations (see Chapter 7, Learning), refactoring (Chapter 8, Abstraction), and retrospectives (see Chapter 11, Reflection).

Tasks

1. For each of the above mechanisms, explain it can be conceived as a knowledge-gaining device that helps cope with changes in software requirements.
2. What other agile practices can be considered knowledge-gaining devices that help cope with changes in software requirements? Explain why.
3. Analyze the notion of change in software requirements within the HOT—Human, Organizational, and Technological—analysis framework.

12.4.2 Organizational Changes

In Chapter 4, Time, we met one characteristic of agile software development—tightness (Hazzan and Dubinsky 2005). The concept of tightness gets its importance, among other reasons, from the fact that software development processes are characterized by changes. A tight process keeps development on track, and if it gets off track, the method's relevant practices return it to the right path. These practices are based mainly on the ongoing feedback that tight software development processes provide. If these practices are rarely applied, that is, if the methodology is not tight, the chances to get off track increase. Since agile software development is a tight process, it leads the agile team onto safe ground, even though changes are introduced along the way. In addition, tightness keeps the development rhythm; this characteristic is important since, once again, software is an intangible product, and therefore a tight process to follow and support its development is needed.

In practice, the tightness of agile software development influences the daily activities of all the project participants—the team as well as the other project's stakeholders, as is illustrated in what follows.

First, agile development inspires cooperation. Among the many implications of this fact, this means that knowledge should be shared among all the projects' stakeholders and that the development environment should encourage each practitioner to contribute his or her professional knowledge. This is achieved by the transparency that characterizes agile environments. This transparency is expressed among other means by the development environment itself, the practice of whole team, and the ongoing collaboration with the customer (see also Chapter 9, Trust). The atmosphere that such a development environment establishes contrasts with development environments in which practitioners tend to conceive of knowledge as power, and accordingly tend not to share their knowledge.

Second, the agile approach delivers a new message with respect to the customer role in the development process that team members should be aware of and act accordingly. Specifically, the agile approach legitimizes the customer's gradual understanding of the requirements and inspires listening-to-the-customer processes in order to understand the customer's needs. The shift in customer perception introduced by the agile approach changes the customer-teammate relationship and replaces the common belief among team members that customers do not know their requirements (see Chapter 3, Customers and Users).

Third, work habits change in agile software development. Among the different changes, we mention the responsibility inspired by the agile approach, the well-planned development process that results in a sustainable pace (see Chapter 4, Time), and the development environment itself, which is open and transparent to everyone. When the agile approach is introduced in an organization, these work habits should replace different work habits, such as long work hours, that result mainly from an unplanned development process and that may reduce quality.

Fourth, the agile approach aims at enabling all the practitioners who participate in the development process to communicate as much as needed. Such a collaborative working environment requires that all practitioners will be in that environment at a certain period of time; it is not supported in organizations in which practitioners arrive and leave the workplace whenever they wish. This last described behavior, which does not support communication, might be the result of the belief that since an intangible product is being constructed, i.e., software, no dependencies exist between practitioners who work on the same project. However, as has been acknowledged here, this very reason—the fact that software is an intangible product—requires *huge* and *strong* dependencies between all the project stakeholders. In practice, some agile activities, such as the stand-up meeting, *require* all team members to attend the development site at the same time and place.

Task

How are the above characteristics of agile software development connected to the Agile Manifesto (see Chapter 1, Introduction to Agile Software Development)?

In what follows we use Plotkin's framework for dealing with change for the characterization of a transition process to agile software development. We focus on the initial stages of the change required when an organization wishes to start the transition to agile software development. As we shall see, such a transition has agile characteristics by itself. An agile approach towards a transition to the agile process implies that different agile practices are applied with respect to the transition process, such as tracking, stand-up meetings, short iterations, customers on site, and role assignments.

Task

For each of the above mentioned agile practices, explain how it is compatible with Plotkin's framework for coping with change.

12.4.2.1 Reducing the Change Scope—Space and Time Reduction

If a decision is made to transition to the agile process, a team should be selected to start the transition. The decision to begin with one team, which is organized in one development space, is a good illustration of scope reduction in terms of space.

This selected team should learn agile software development. One way to start this learning process is to apply several agile practices right at the beginning of the transition process, and gradually add agile practices to be applied by the team. One practice that is suitable to start with is agile planning. The gradual application of agile software development should be discussed with the team based on reflection and retrospective sessions (see Chapter 11, Reflection).

Another option by which it is possible to start the learning process of agile development is to let the team learn the basics of agile software development in a condensed workshop. A suggestion for such a condensed workshop, which lasts two days, is presented later in this chapter. It is preferable not to mange this workshop in the team's development environment, but rather in a learning site organized for the purpose. In this workshop, the team learns the basic knowledge required to start the change process and the implementation of agile software

development. Based on this knowledge, as the team proceeds with the application of the agile process, it continues the ongoing learning of agile software development.

Task

Does the condensed workshop reflect scope and time reduction? Why?

In addition, as part of the transition process to agile development, a time line for the *transition process* is set, exactly as it is set with respect to agile *software projects*. Thus, for example, short iterations are set which aim at helping the management clarify its requirements and expectations of the transition process and define measures to navigate that process (see Chapter 5, Measures).

Task

For each of these two practices—short iterations and measures—when applied to the transition process to agile development, explain:

- How does it reflect an *agile transition process* to agile software development?
- Lay out the details of its implementation with respect to the transition process to agile development.
- In what ways are these practices compatible with Plotkin's framework?

12.4.2.2 Join the Change—Diversity

In the transition process, roles are assigned to team members, and the customer for the transition process is specified. Measures are set to reflect the concerns of *all* the project stakeholders. Several of the reflective sessions in the iteration summary meeting (see Chapter 3, Customers and Users) are dedicated to the transition to agile development (and not on the agile software development itself). All these means enable the participants to express their feelings, suggestions, etc., and consequently to support the transition process.

Task

How does each of the practices mentioned above reflect diversity?

12.4.2.3 Join the Change—Knowledge-Gaining Devices

Among the main knowledge-gaining devices used in the transition process to agile development we mention reflections and retrospective sessions (see Chapters 7,

Learning, and Chapter 11, Reflection). These processes are dedicated not only to improving the software development process, but also to supporting the transition process to agile development that takes place in the organization. Based on lessons learned in these reflection and retrospective sessions, the continuation of the transition process to agile development is determined, and issues such as scalability (see Chapter 2, Teamwork) are considered.

Tasks

1. What other knowledge-gaining devices can be applied with respect to the transition to agile software development?
2. Analyze the notion of organizational change within the HOT—Human, Organizational, and Technological—analysis framework.
3. Summarize how Plotkin's framework is applied to change introduction in software requirements and to organizational change.

12.5 Transition to an Agile Software Development Environment

Since, as we have seen, agile software development introduces changes in work habits, when an organization wishes to transition to agile software development, an organizational change is required.

Different approaches are suggested for these organizational changes in general and for transition processes to agile software development in particular. For example, Manns and Rising (2004) suggest 48 patterns for change introduction. Christensen et al. (2006) propose four major categories of cooperation tools in change processes: power, management, leadership, and culture. Choosing the right tool, according to the authors, requires assessing the organization along two critical dimensions: the extent to which people agree on what they want, and the extent to which they agree on cause and effect, or how to get what they want.

Task

Review the frameworks for transition processes suggested in the above resources. How are they compatible with the agile approach?

The notion of agile software development is usually introduced into an organization by one of the practitioners in the organization. This person can

come from any of several different hierarchical levels. Sometimes this person is a developer who feels that something is wrong with the way software is currently developed in his or her team or in the organization. Sometimes, this person may be one of the team leaders who feels that he or she does not control the process he or she is in charge of, and is brave enough to admit it. In yet other cases, the notion of agile development is raised by a person with a higher position in the organization who is not satisfied with the results of the current software development process, hears customers' complaints, and understands the financial implications of this dissatisfaction. In most cases, many informal conversations take place before the organization takes any action to start the transition process to agile development.

Task

Can you suggest what topics are discussed in these informal conversations? What role do they play in the transition process to agile software development?

After the decision is made by an organization's management to examine the suitability of agile software development for the organization, in many cases an outside consultant is hired to assist and support the transition process. The position of the consultant is important, because he or she will not be involved in the organization's politics and socialization and can see and analyze the development process and the transition to agile development from a more objective viewpoint. The role of the consultant in the transition process to agile development is further elaborated in Chapter 13, Leadership.

12.5.1 Organizational Survey

One possible way by which the consultant can learn about the organization is through an organizational survey, whose purpose is threefold: First, the survey may help the consultant to understand the current situation and status of software development in the organization. Second, the survey helps the consultant become familiar with the organization's terminology; the consultant's familiarity with this terminology will assist the assimilation process, if it is decided to transition to agile software development. Third, a survey may help in the decision as to whether the agile approach fits for the organization at all; if it is found that agile software development might fit the organization, the data gathered during the survey can be used for the decision on how to initiate the transition process to agile software development. Based on such a survey, however, it can also be found that either the agile process does not fit the organization or that a change is not needed at all for the organization.

In what follows we suggest an organizational survey process carried out by a consultant who is hired by a software organization to check the suitability of the agile approach for its software development process.

Stage 1. Meeting(s) with the people who initiated the transition process to agile software development. The target of this meeting(s) is to clarify the organization's needs, set mutual expectations, describe the nature of the organizational survey, and explain the process that will follow the survey, if it is decided to start a transition process to agile software development. In this meeting(s) the consultant usually meets the people who will control and supervise the transition process, if it is decided at all to transition to agile software development.

Stage 2. Two days of interviews and meetings with people from the organization who hold different roles and belong to different levels and teams. The main purpose of these interviews is to get as diverse, multidimensional, and wide a picture as possible about the current situation of software development in the organization. If possible, it is recommended that people from outside the organization be interviewed also.

Each interview/meeting starts with a short description of the goal of the interview, its structure, and its length—one hour. The interviewee is assured that the information he or she shares will be used only for the survey purposes and will be kept anonymous. Then the practitioner works on short and quick questionnaires for about 15 minutes. The idea is to guide the practitioners through a reflective mode of thinking (see Chapter 7, Learning, and Chapter 11, Reflection). These questionnaires should be adjusted for each organization according to its nature and specific characteristics.

Next, an open interview takes place in which the practitioners are asked to reflect on and describe their software development experience in the organization. The interview is guided by questions such as: Describe the development process that is currently applied in your organization. What is your role in this process? What are the process's benefits? What are its pitfalls? Describe a successful story of software development by your team. Describe a story that shows the failure of a software project that your team experienced. How would you improve the development process of your organization? How would you describe the knowledge flow and sharing in your team? In your project? In the organization?

These questions usually lead to follow-up questions about the development process in the organization. Additional questions can be asked, of course, if needed.

Stage 3. Data analysis. The data gathered during these two days are analyzed by the consultant. The analysis findings are used first for an examination of the current software development status in the organization, and second, to determine whether the agile approach for software development fits the

organization. If it is decided to transition to agile software development, the survey's results guide the selection of the team which will start the transition process.

Stage 4. Presentation of the survey results to the organization's management. After the data are analyzed, a meeting with the organization's management takes place, in which the survey findings are presented by the consultant, a written report is submitted, and the continuation of the assimilation process is determined.

In the report (both written and verbal) it is recommended to start with some details about how the survey was conducted (when, how, who was interviewed). Then the focus should be placed on the main observations (high level description first, details latter; see chapter 8, Abstraction), the benefits of the current development process in the organization, the pitfalls/problems of the current development method, and, of course, general thoughts about the process continuation. The conclusion should summarize in a high level description what has been presented. It is recommended that the human, organizational, and technological (HOT) aspects be addressed in the presentation and in the written report, in order to provide a wide and comprehensive picture as possible of the development process that takes place at the organization.

During the presentation it is important to stop from time to time to hear the management's reaction and to answer questions if raised. It is reasonable to assume that if the survey's findings are not positive, the management representatives will express some resistance, and sometimes even anger. Such reactions should be understood and can be approached in different ways. For example, it can be recommended that they take advantage of this opportunity to learn about the opinions of the practitioners who work in the organization with respect to their working environment in general and their software development process in particular.

Task

How is Plotkin's framework for coping with change expressed in the way an organizational survey is conducted?

12.5.2 Case Study 12.1. A Report of an Organizational Survey

This case study presents the report of an organizational survey conducted in a large company whose management decided, based on the survey results, to start implementing the agile approach. As just described, the written report was

submitted together with a presentation to the organization's management. We should mention that many of these findings are not unique to this company and have emerged in other organizational surveys.

Company A—Organizational Survey about Software Development Processes

The survey is based on:

- three meetings with the organization's management at the company sites;
- interviews with about 20 practitioners with different roles in the organization;
- questionnaires which were filled in by about 30 practitioners in the organization.

The report focuses on different aspects of the development process and is divided into two parts: findings and recommendations. Each of these parts addresses four aspects: customers, development methods, development culture, and teams.

Findings

Customers

- Customer perception: Customer satisfaction is considered one of the main parameters for software quality. At the same time, however, when customers are confident with respect to their requirements, resistance is raised.
- Only a few team members are in contact with the customers—usually team leaders and system analyzers. At the same time, however, contact with the customers is desired by all the practitioners, while code writing and tests are less attractive activities.
- The need for a defined working process with the customers is expressed by different practitioners.
- Requirements: Problems in requirement understanding, incompatibility with the customer's requirements, assumptions added to the customer requirements, ongoing interaction with the customers not taking place, developers humiliated in front of customers when it turns out that what they have developed doesn't fit the customer's requirement.
- Contracts: A need for a change in the contracts signed with the customers is expressed.

Development methods

- Absence of a defined development methodology.
- Absence of development resources.
- Testing: No automatic tests, absence of an ordered testing process, many bugs, testing schedule is not met.
- Specification: The specification definition process sometimes takes too long without coding, sometimes specifications do not exist at all, in other cases there are failures in the requirement formulation and in the design.
- Changes: Absence of a methodology for change introduction, changes cause delays, it is unknown what the number of changes is.
- Absence of measures.

Development culture

- Time management: Wasted hours and unexploited time, at the same time—short schedules are promised to the customer.
- Problem in knowledge and information maintenance, management, and sharing.

Teams

- Team leaders have too many roles.
- Developers are evaluated by the number of bugs they fix.
- Roles are not clearly defined.
- Developers are overloaded.
- Importance is attributed to reflective processes.

Recommendations

Note: Most of the following recommendations have been suggested by the practitioners who participated in the organizational survey.

Customers

- Set a responsible and reasonable development process.
- For projects with users—build a user forum.

- For each project, define the customer and increase the customer involvement in the development process.

Development methods

- Establish a development method that encourages an organized work process.
- Define a reasonable schedule.
- Assign leadership with responsibility and professional experience.
- Establish a mechanism for code quality review.
- Improve the test quality.
- Integrate reflective meetings on a regular basis.

Development

- Ensure organized and fair work distribution among the developers.
- Establish a culture of adhering to deadline/scope/quality.

Teams

- Improve cooperation between teams as well as communication among teammates.
- Define measures for team and individual evaluation (not necessarily by the number of bugs).
- Grant responsibility to team members for the product quality.
- Establish an organized process of information sharing.

General Message

Management commitment to the development processes should be reflected by:

- development policy;
- development processes (including measures);
- project transparency to all project stakeholders, mainly developers and customers.

End of Organizational Survey

Tasks

The following tasks address Case Study 12.1.

1. What can be learned about the organization in which the above organizational survey was carried out?
2. Based on the report, can you describe the current development process applied in the organization?
3. How would you address each of the recommendations suggested in the report?
4. Would you recommend that the organization's management start a transition process to agile development?
5. In your opinion, how did the organization's management react to the fact that most of the recommendations were suggested by the practitioners who participated in the organizational survey?
6. Can you speculate which items in the report would have influenced the management of the organization to decide to start a transition process to agile development?
7. Analyze the writing style of the report. In your opinion, why was this style chosen?
8. How is the HOT—Human, Organizational, and Technological—analysis framework for the software development environment expressed in this report?

12.5.3 Case Study 12.2. Applying an Agile Process to a Transition Process

This case study illustrates how agile practices can be applied for non-software projects in this case, for the tarnation to new software development process. For further elaboration on this idea, see Chapter 10, Globalization.

Sharon was in charge of the development of a software system used by thousands of users and developed by 60 highly skilled developers and testers. The 60 practitioners were organized in a hierarchical structure of smaller teams. To foster communication, we also use the term *team* when we refer to the group of 60 practitioners.

Sharon was assigned by the organization's management to change the current development process to one that would enable a rapid response to customers' change requests. Sharon had one year to implement this change. In brief, Sharon's task was to lead an organizational change the highlight of which would be the adoption of an unknown (in the beginning) software development process.

The management supported this process and specifically declared that while reduction in functionality might be accepted, quality and fitness to customers' needs would not be compromised.

Since Sharon's team was big, such a change could not be implemented at once. Rather, it would need to be performed gradually and to be monitored and planned accordingly.

Sharon asked two representatives of a consultancy company to help her plan the transition to the new development process (that had not been defined yet). In other words, Sharon asked them to help her plan a *non*-software project that aimed at the assimilation of a new (as yet unknown) software development process. This process of assimilation should consider the individual's interests and the resistance each individual and party might raise, as well as the harmony and synergy between the different changes that would take place with each team in each of the process phases.

Sharon started the actual work by meeting with the consultants. When Sharon described the process according to which the team currently worked, the following basic problems were identified:

- Too long a feedback process: In some cases it took a year and a half to get feedback from users who had asked for a change. In many cases no feedback was received from the users at all. In yet other cases when feedback was received and addressed, it was no longer relevant.
- Only a few people communicated directly with the customer; developers did not have any contact with the users.
- Intercommunication between team members was cumbersome.
- Each developer worked at one abstraction level: Developers' roles were very narrow and forced them to stay at one specific abstraction level (such as code level or design level) during the entire development process.
- Too big (75%/25%) a nondevelopers/developers ratio: more specifically, only small number of people actually wrote code while all the others worked on requirement gathering, design activities, etc. This structure was, in fact, part of the one abstraction level problem described before.

- Resistance to change: The source of this resistance was people's worries about losing some of the authority they had in the current organizational structure. The team members, who did feel a need for change, were too low in the hierarchy, and therefore their ideas were easily blocked by their superiors.
- No personal responsibility: In other words, it was not clear who was in charge of what.
- Lack of time to deal with all these problems.

As a preparation for this transition, a task force of 10 volunteers was established, whose aim was to formulate and plan a process at the end of which the entire team would work according to the new yet-unknown software development process.

The task force was composed of representatives from all team levels. It met every two weeks and worked according to the following agile practices. (Note that the agile approach was applied for a planning project and not for a software development process.)

- Small versions and iterations were defined.
- The customer's side (Sharon) was constantly represented.
- Customers' stories were gathered and divided into iterations.
- Customers' stories were broken down into specific tasks.
- Tasks were assigned at each meeting to each of the task force members. The members estimated the time needed for the task performance.
- Stand-up meetings took place at the beginning of each meeting.
- Special roles were assigned to the teammates in order to control the process.
- Tests were defined in order to check whether the customers' stories were performed as required.

Thus, for example, a planning session took place in the first meeting, when Sharon played the customer of the transition project, and the different assignments (such as reading and learning relevant material, mapping the project's current organizational structure, etc.) were allocated to the task force members for the first two weeks' iteration.

The second meeting, which took place after two weeks, started with a stand-up meeting in which each member of the task force reported on the tasks he or she had accomplished during the past two weeks. Then the following roles were assigned: coach, tracker, on-site customer, and those in charge of acceptance testing, presentations, documentation, design, and code effectiveness and correctness. Since

the project was a planning project and not a software project, the roles were defined slightly differently from their description in Chapter 2, Teamwork.

Next, the task force members presented their first iteration products on which they had worked during the previous two weeks. Then a planning session for the second iteration of the first release was facilitated. The meeting ended with a retrospective session of 20 minutes (see Chapter 11, Reflection), in which topics such as the following were discussed: the increased ability to cope with many tasks if they are planned, allocated, and carried out properly, and the fitness of this mode of work for the task force. The first voices of resistance were heard.

So far three teams of this projects had started working in accordance with an agile process, and scalability issues started to be discussed at the project level.

Tasks

1. Analyze Case Study 12.2 using Plotkin's framework of coping with change.
2. Analyze Case Study 12.2 using the HOT—Human, Organizational, and Technological—analysis framework.

12.6 Change in Learning Environments

This meeting is part of the development phase of the third iteration. As with meetings 9 and 10, which were dedicated to the development of the second iteration, the academic coach can participate in this meeting and take the opportunity to assess the project's progress at both the team and individual levels.

12.6.1 Introducing the Teaching of Agile Software Development

We use Plotkin's framework for showing how the teaching of agile software development can be introduced into academia. It is relevant for cases in which a department wishes to let its students experience the development of projects in team of 10–12 students in an agile process, as is described in this book.

12.6.1.1 Reducing the Change Scope—Space Reduction

Start the assimilation process with one course. Dedicate a development workspace, such as the studio, for each student team.

As presented in Chapter 1, Introduction to Agile Software Development, it is advisable, in order to ease the academic coaches' adoption of the new coaching environment, to offer in the first semester of the transition projects that had already been developed in previous semesters and to adjust the scale of such projects to team sizes of 10–12 students. This way the academic coaches can focus on the new coaching environment without being distracted by technical issues. In the following semesters, when the coaching team feels more comfortable with the framework, new topics for projects can be introduced.

12.6.1.2 Reducing the Change Scope—Time Reduction

In an academic environment, the implementation of this idea is immediate and natural, since semesters determine the time framework. As is illustrated in this book, the semester is divided into three development iterations.

12.6.1.3 Join the Change—Diversity

Form diverse student teams, since it is more likely that in diverse teams some of the students will like the change. The same idea should be applied as well for the selection of the team of the academic coaches.

12.6.1.4 Join the Change—Knowledge-Gaining Devices

Encourage ongoing reflective processes (on the individual level and the team level) both with the students and with the academic coaches. In addition, share with the outside environment what goes on in the transition process, listen to its reaction, and navigate the assimilation process accordingly.

Additional details about the transition process to agile software development as it takes place in academia can be found in Hazzan and Dubinsky (2003) and in Dubinsky and Hazzan (2005).

12.6.2 Two-Day Workshop

Agile software development can be introduced in a condensed form of a two-day workshop. After the general description of the workshop, we describe how the workshop can be adjusted for the introduction of agile software development to a group of academic coaches.

In the two-day workshop presented here, agile software development is introduced while enabling the participants to experience as far as possible the actual construction of a software system. The idea is to show the participants just how much can be achieved in two days if the planning and time management are conducted properly, while paying attention to the customer's needs. In order to achieve this goal within the time limit, during the workshop the participants complete the development of the first iteration of a software system of their choice. In addition, reflective processes are conducted, and a discussion is held on the suitability of agile software development for the organization in question.

The pedagogy that guides this workshop is the same pedagogy reflected in the teaching and learning principles introduced in the different chapters of this book (for a complete list of these principles, look at Chapter 14, Delivery and Cyclicality). There are, however, several additional useful teaching principles which are especially relevant for short presentations of agile software development such as this two-day workshop.

Group size. The preferred group size for this workshop is 6–12 participants. The upper limit (12) is defined to reflect realistic team sizes in the software industry. The lower limit (6) is required because this number of participants provides a feeling of how agile development works for a team. In addition, all participants are required to attend the full two days. If a participant is forced to leave the workshop for some unexpected reason, it is a good opportunity to discuss the analogy of such a case to real life situations.

Sustainable pace. It is important to make sure that the development performed during the workshop is conducted at a reasonable pace. The fact that the first iteration is developed within two days serves to reinforce the participants' feelings about the potential contribution of the agile approach to the software development process.

Workshop site. The host organization should equip the room in which the workshop is to take place with a large table for the planning session, computers, flipcharts or a whiteboard, and a coffee corner.

Narrative. In such short term presentations, it is *not* sufficient to merely describe those practices which there is no time to experience. As an alternative to a technical explanation of these practices, a storytelling approach, which has become popular in management process, can be adopted. In this spirit, the agile practices (in various levels of detail) can be presented using the following story.

The story describes two days in the life of a software team member working in an agile environment: one Business Day (see Chapter 3, Customers and Users) and one development day. In the course of the story, the audience is invited to envision themselves in the environment described and to compare it continuously with their own current software development environment. As in a real story, in

Table 12.1 Schedule of the first day of a two-day agile workshop (© [2003] IEEE)

Time	Activity
10:00–10:30	Introduction
30 minutes	The day starts with the presentation of the two-day story and references are made to what the participants can expect to experience during the next two days
10:30–11:00	Selection of a topic for the project
30 minutes	The participants decide on the software project they are going to develop in the workshop. The idea is to show that any project can be developed using agile software development
	The workshop participants are asked to suggest ideas for the software project and then take a vote on the preferred topic. The person who suggested the selected topic becomes the main customer (see Chapter 3, Customers and Users) in the later stages. In case of disagreement among the audience with respect to customer stories' preferences, he or she will decide on these preferences
11:00–13:00	**Planning session**
2 hours	All stages of the planning session (see Chapter 3, Customers and Users, and Chapter 4, Time) are performed: telling of customer stories, allocation of releases and iterations, simple design of the first iteration, breakdown into development tasks, allocation to developers, time estimation, and load balance. During all of the stages, rather than explaining the rules of the game up front, they are introduced during the session itself (see Teaching and Learning Principle number 3, Explain while doing). Such an approach enables the participants to immediately apply what they hear. Additional guidelines used during the planning session include:
	Telling customer stories
	The participants are encouraged to add stories. Since most of the participants have no previous experience being a customer, some feel uncomfortable with this request. When participants are encouraged to tell stories and they do start to add stories, they may begin to sympathize with this task, which customers are usually asked to perform
	Distinction between the customer and the end user is made (see Chapter 3, Customers and Users). Sometimes, perhaps due to lack of experience, participants confuse these two roles. This stage of the planning session offers an appropriate opportunity to distinguish between the two roles
	Role assignments
	The roles presented in Chapter 2, Teamwork, are assigned for the entire duration of the iteration. Specifically:
	Customer: It is important that the role of the customer is not played by the workshop's facilitator
	Team Coach: This role, too, must be played by one of the workshop participants and not by the workshop facilitator. The team coach is in charge of the actual development of the software, as in real agile teams; whereas the workshop facilitator is in charge of managing the workshop itself.
	The first iteration
	After the allocation of customer stories into releases and iterations, the focus is placed on the first iteration. With respect to this iteration, the following issues are discussed with the participants:
	the need to split a customer story;

Table 12.1 (continued)

Time	Activity
	the importance of clarifying with the customer topics that appear to be unclear;
	the importance of documenting the process that takes place during the planning session (decisions, questions, clarifications, etc.); and
	the role of the whiteboards as a means of communication.
13:00–13:30	Lunch
30 minutes	
13:30–14:00	Discussion of problems using metaphors
30 minutes	Metaphors can be used in order to increase the participants' awareness of problems that may arise during the process of software development (see Chapter 3, Customers and Users). Metaphors can be used also in order to improve the team members' understanding in cases in which they are relatively familiar with developed applications
14:00–15:00	Planning session (Cont.)
1 hour	Delve into the development tasks
	Time estimation
	When participants are asked to estimate the development time for each task, it is important to legitimize time for learning the material needed for the accomplishment of the tasks in the first iteration
	Participants' awareness should be increased with respect to the learning value of time estimation
	An explanation of how testing (see Chapter 6, Quality) is dealt with in the time estimation should be presented
	Total development time
	The total estimated development time is summed up by one team member. Participants get a better sense of this idea when it is done by a participant rather than by the workshop facilitator. A factor of 1.5 is introduced into the calculation to account for pair programming
	The time available during the condensed format of the workshop is calculated. For example: 8 people $\times$ 5 development hours in the second day = 40 development hours. This number (40) is compared with the total number of hours estimated for the development tasks of the first iteration; stories are added or removed accordingly
	The concept of load balance is addressed. It is important to ensure that for such a short workshop the difference in loads between team members does not exceed 1.5 hours
15:00–15:45	**Acceptance tests**
45 minutes	Acceptance tests are discussed in small groups and the set of acceptance tests are determined by the entire team
15:45–17:00	**Reflection on the first day**
75 minutes	The day ends with a retrospective session on the first day (see Chapter 11, Reflection). Specifically, it is stimulating to perform this retrospective by addressing the idea of "changing hats" (team members, learners of a new software development method, customers) and the main questions that arise with respect to each hat
	In preparation for the second day of the workshop, participants are asked to start thinking about agile-related topics that in their opinion can be implemented in their own organization

Table 12.2 Schedule of the second day of a two-day agile workshop (© [2003] IEEE)

Time	Activity
10:00–10:15 15 minutes	Stand-up meeting. The participants are requested to explain in about three sentences what they learned in the first day of the workshop and to share one of their ideas regarding the implementation of the agile approach in their workplace
10:15–10:45 30 minutes	**Testing in agile software development** Short explanation and demonstration of test-driven development (TDD). See Chapter 6, Quality Discussion about the benefits and pitfalls of TDD
10:45–13:00 2 hours and 15 minutes	Development of the first iteration. At this stage, the team coach enters the picture. It should be clear to the participants that they are responsible for making relevant decisions and completing the development of the first releases by the end of the day
13:00–13:30 30 minutes	Lunch break
13:30–15:15 1 hour and 45 minutes	Completion of development and integration.
15:15–15:30 15 minutes	Presentation of first iteration. If the first iteration is ready, it is presented to the "customer" and to the workshop facilitator. If the first iteration is not ready, the opportunity is taken to discuss the reasons that led to this situation. It is also a good opportunity to conduct code review and address related topics, such as coding standards and refactoring
15:30–17:00 1.5 hours	Reflective session. In this stage, participants are asked to suggest topics and activities they performed well, and activities that they feel they could have performed better. The topics are listed on the board in two columns: human- and organizational-related topics and technically-oriented topics. The participants are not yet privy to the classification criteria. When no more new ideas are suggested, the participants are requested to offer titles for the two columns. Usually, at some stage, the above categorization is suggested. One clear lesson that emerges from this session is that the main issues are related to human, technical, and organizational aspects of software development, and not to technological aspects only. Table 12.3 illustrates one outcome of such a reflection process Implementation of agile software development in the participants' organization. This last part of the workshop is dedicated to a discussion in which the participants present their thoughts on the implementation of agile software development in their organization in general, and in their team in particular

addition to presenting the agile software development by means of the main character's actions, it is advisable to spice up the story with various details (such as a description of the site where the story takes place—the setting of the tables and the computers in the development environment, the information posted on the whiteboards, etc.).

Table 12.3 Example of the reflection stage conducted at the end of the second day of an agile workshop (© [2003] IEEE)

	Human and organizational topics	Technological topics
Activities we were good at	Establishment of a positive atmosphere Team work Learning Cooperation Working in pairs	Java Software development Test-first
Activities we can improve	Team management Additional learning	Software detailed-design Coding while relating to design

Tables 12.1 and 12.2 present the schedules for the two-day agile workshop. In order to illustrate how the days look in practice, hour notations are added to the workshop plan.

As can be observed from Table 12.1, the main role in the first day is that of the customer and the message to be conveyed in this day is that the customer is an integral part of the development environment.

Prior to the beginning of the second day of the workshop, the customer stories are presented on the whiteboard according to the releases and iterations that were determined the previous day.

When the participants enter the workshop room at the beginning of the second day, this presentation illustrates very clearly how the whiteboard can serve as a communications tool. The agenda for the second day is also presented on the whiteboard.

Table 12.2 presents the schedule for the second day of the two-day agile workshop, together with an explanation of the main activities conducted during each time slot. The main role in the second day is that of the coach.

12.6.3 Two-Day Workshop Format for a Team of Academic Coaches

When the two-day workshop is conducted with a team of academic coaches who are not familiar with agile software development, but need to start guiding students in the software development process using the agile approach, several additional topics can and should be discussed during the workshop. For a comprehensive explanation of such a workshop, see Dubinsky and Hazzan (2003).

Telling customer stories. Discuss with the academic coaches the fact that in educational environments some students tend to refrain from adding

customer stories, since they assume they will have to develop all of the suggested stories. In such cases, it is important to explain to the students that the final decision about which stories will be introduced into each iteration is made according to the available time and the number of students on the team. Accordingly, the students should receive a clear message that they need not worry about adding customer stories.

Student teamwork. Discuss with the participants the "free riders" phenomenon and ways in which it is possible to grade projects so as to overcome this problem. One suggested solution is to establish a grading policy that takes into account both the personal contribution to the software project as well as the team achievement (see Chapter 2, Teamwork).

Reflective sessions. In the pedagogically-oriented workshop, it is important to address also pedagogical topics such as learning through activity (see Chapter 7, Learning), awareness of metaphors (see Chapter 3, Customers and Users), students' self-learning of new topics, and knowledge sharing among team members.

12.7 Summary and Reflective Questions

1. Review special change-related events that occurred during the software development process in the studio during the semester. Can they be considered as successes? As failures? Can they be explained in terms of Plotkin's framework of coping with changes?
2. In your opinion, is an evolutionary perspective on software projects useful? In what sense? How might an evolutionary perspective help improve software development processes?
3. Examine the course structure (taught by this book) within Plotkin's framework. With what changes does the course deal? How does it do that?
4. Explore the connections between the ideas presented in this chapter and Section 5: Management, of the Code of Ethics of Software Engineering, presented in Chapter 9, Trust.
5. Construct two case studies about software development projects that illustrate Plotkin's framework for coping with change—one demonstrating a successful case; the other demonstrating a case in which a transition to the agile process did not take place or failed. You can look for such case studies in the software engineering literature.

12.8 Summary

One of the main ideas of this chapter is the legitimization of agile software development for change introduction. We use Plotkin's framework for coping with change to explain the agile approach towards change. We delve into the details of the transition process to agile software development, reviewing how an organizational survey can be conducted, describing two case studies about the transition to agile software development, and outlining a two-day workshop on agile software development.

References

Christensen CM, Marx M, Stevenson HH (2006) The tools of cooperation and change. Harvard Bus Rev 84(10):72–80, 148

Dubinsky Y, Hazzan O (2003) eXtreme programming as a framework for student-project coaching in computer science capstone courses. Proceedings of the IEEE international conference on software—science, technology & engineering, Herzelia, Israel, pp 53–59

Dubinsky Y, Hazzan O (2005) A framework for teaching software development methods. Comput Sci Educ 15(4):275–296

Hazzan O, Dubinsky Y (2003) Teaching a software development methodology: the case of extreme programming. The proceedings of the 16th international conference on software engineering education and training, Madrid, Spain, pp 176–184

Hazzan O, Dubinksy Y (2005) Clashes between culture and software development methods: the case of the Israeli hi-tech industry and extreme programming. Proceedings of the agile 2005 conference, IEEE computer society, Denver, Colorado, pp 59–69

Manns ML, Rising L (2004) Fearless change: patterns for introducing new ideas. Addison-Wesley, Reading, MA

Plotkin H (1997) Darwin machines and the nature of knowledge. Harvard University Press, London

13
Leadership

Abstract

Leadership is the ability to influence people, encouraging them to behave in a certain way in order to achieve the group's goals. Leadership is independent of job titles and descriptions; usually, however, in order to lead, leaders need the power derived from their organizational positions. There are different leadership styles, such as task-oriented versus people-oriented, directive versus permissive, autocratic versus democratic. While leaders can shape their leadership style according to circumstances, followers might prefer different leadership styles depending on their situation. Agile software engineering adopts a leadership style that empowers the people involved in the product development process. For example, instead of promoting the idea that "leaders should keep power to themselves in order not to lose it," the agile approach fosters the idea that "leaders gain power from sharing it." This idea is expressed, among other ways, by the transparency of the agile development process that makes information accessible to anyone and enables each team member to be accountable for and fully involved in the development process.

13.1 Overview

Leadership is a social phenomenon required for achieving a group's goals (Nirenberg 2002). Various definitions for leadership exist. The evolution of these definitions across the last century reflects a change in leadership conception:

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_13, © Springer-Verlag London Limited 2008

from leadership as a unilateral ability or process to control people, to leadership as an influential relationship between leaders and followers (Ciulla 1998, Cooper 2005). In other words, leadership definitions have moved from the description of a unique person who was born a leader, to comprehensive definitions for leadership that refer to all people involved—leaders and followers, the interaction among them, and the task in hand (Topping 2002).

In software development environments, leadership is required for the management of different activities in different situations. *Management* of a software project includes team management, process management, quality management, and cost management (Hughes and Cotterell 2002, Mayrhauser 1990, Sommerville 2001, Humphrey 2000), and consists of constant assessment of its own activities (Putnam and Myers 1997, Pulford et al. 1996). Table 13.1 outlines these four categories; for each category several main activities are described. For example, time management is part of process management. As discussed in Chapter 4, Time, time is managed on different levels of the organization and for different process phases. Specifically, one can manage his or her personal time within a day or within a specific iteration/release; a team manages its time within iteration/release; and a team that is composed of several teams should also manage and coordinate its time within iteration/release.

In this chapter we explore leadership and leadership styles and examine how leadership is expressed in agile teams. Since leadership is not just an expression of a position in a hierarchy or a chain of command, project and team managers are

Table 13.1 Software development management

Management category	Description
Team management	Increasing awareness of human factors
	Consolidating group and introducing organization values
	Arranging suitable development environment
	Updating organization and group knowledge base
	Enhancing communication
	Defining and implementing training programs
Process management	Detailing process activities
	Managing time
	Implementing a role scheme
	Obtaining metrics regarding time, resources, and events
	Improving the process continuously
Quality management	Defining quality goals
	Defining and implementing quality procedures
	Setting metrics regarding quality
	Improving quality procedures
Cost management	Defining and implementing a cost model
	Referring to project, product, resources, and personnel aspects
	Using a costing model as a metric for process status

expected to be leaders and to coach their teams. We describe how agile development methods refer to coaching in general and describe a case study about academic coaching in particular.

13.2 Objectives

- Readers will become acquainted with the concept of leadership in general and leadership styles in particular, and with how leadership is expressed in agile software development environments.
- Readers will become acquainted with leadership levels and their impact.
- Readers will become familiar with the leadership role of the methodology change leader.
- Readers will learn the practice of coaching agile teams.
- Readers will learn to assess leadership expressions in their environment and evaluate leadership contributions to teamwork.
- Readers will gain the skills to evaluate how different agile notions and practices enhance leadership.

13.3 Study Questions

1. What is leadership and how can its impact be measured?
2. Identify two leaders in your development environment. Why do you perceive them as leaders? What do they lead? In general, what characteristics do you use to identify leaders?
3. Based on your experience, what is expected from a team leader? For each of these expectations, explain how the team leader should handle it and what difficulties he or she may encounter in this process. Base your analysis on examples of leadership styles you are familiar with.
4. Describe the coaching characteristics desired from an agile team leader. What are the two characteristics you like the most? Why? What are the two characteristics you dislike the most? Why?
5. Based on the previous chapters, describe two leadership mechanisms in agile teams.

13.4 Leaders

Management and leadership are described as a continuous act that aims to keep the balance between characteristics that on the one hand dominate and control, and on the other hand are inspiring and strive for creativity. Table 13.2 (adopted from Huff and Moeslein 2005, originally from Drath 1998) presents the evolution of leadership models, indicating a shift in leadership perception.

Tasks

1. In your opinion, what is the main change in leadership perception across time?
2. What leadership style suits you personally? Why?
3. Select two issues or scenarios in software development environments and analyze them using the evolutionary model of leadership.
4. Select two agile notions and explain them using the evolutionary model of leadership.

In software development environments, "Leadership is generally taken to mean the ability to influence others in a group to act in a particular way to achieve group goals" (Hughes and Cotterell 2002, 222). In Augustine et al. (2005) an adaptive leadership is suggested for agile teams, including first, the ability to adapt to changes (see also Chapter 12, Change); and second, the problem-solving approach that considers all stakeholders to be skilled and valuable, relies on autonomous teams, and minimizes up front planning to be able to adapt to changes (see also Chapter 4, Time, and Chapter 7, Learning). It is important to note that autonomous agile teams also have leaders who work in a leadership-collaboration environment (Cockburn and Highsmith 2001). This means that in many cases team members experience collegial relationships with the team leader, not necessarily hierarchical ones.

Table 13.2 Evolving models of leadership (source: Drath 1998, 408) (Reprinted with permission of John Wiley & Sons, Inc.)

	Ancient	Traditional	Modern	Future
Idea of leadership	Domination	Influence	Common goals	Reciprocal relations
Action of leadership	Commanding followers	Motivating followers	Creating inner commitment	Mutual meaning making
Focus of leadership development	Power of the leader	Interpersonal skills of the leader	Self-knowledge of the leader	Interactions within the group

Referring to Table 13.2, the agile approach fits the "modern" and "future" leadership styles, on which we elaborate in what follows.

With respect to the idea of leadership, the notion of common goals in agile teams is expressed mainly by the customers' ongoing collaboration along the entire development process; and by information transparency, which enables each team member to know what these common goals are and to participate in the planning and presentation meetings related to them. Reciprocal relations relate to high levels of cooperation, confidence, and trust among team members. In Hazzan and Dubinsky (2005), we use game theory to explain reciprocation in software development environments by employing the prisoners' dilemma (see Chapter 9, Trust).

Task

Describe three scenarios that exemplify reciprocation in agile teams. For each scenario explain how reciprocation is achieved and expressed, and evaluate its contribution to the development process.

With respect to the action of leadership (Table 13.2), inner commitment is created and enhanced when using the role scheme, by which each team member has an additional specific role that assists the project leadership (Dubinsky and Hazzan 2006) (see also Chapter 2, Teamwork). Though team members are committed, mutual meaning is still needed, i.e., the commitment should provide a specific relevant and meaningful product.

The focus of leadership development aspect of Table 13.2 shows how the leader's position should be developed to improve leadership. While the three first columns focus on the leader, the "future" column deals with the group and its interactions. As the level of leadership increases, the group interactions lead the team, i.e., the way team members communicate, reflect, and collaborate enables the team to lead itself as if there were no leader, while, in practice, high quality leadership exists.

Task

Explain the way group interactions form the focus of leadership development in the agile environment.

13.4.1 Leadership Styles

There are several leadership styles in software development environments (Bruce and Langdon 2000, Hughes and Cotterell 2002).

The first and most common style ranges from dictatorship or autocracy to democracy. The autocratic leader decides alone, while the democrat leader involves the team members in decision making processes. In between these two poles we have the analytic leader, who makes decisions based on data collection and analysis, and the opinion-seeking leader, who seeks for the team members' opinions in order to make a decision.

The second scale ranges from directive to permissive leadership. The directive leader conducts close supervision of the implementation, while the permissive leader encourages team members to have latitude. This scale also includes the level of delegation a leader uses, i.e., the degree to which he or she delegates tasks to team members.

Another scale is known as task-oriented leadership versus people-oriented leadership. The former relates to the degree leaders are focused on the task in hand; versus the latter, which relates to the degree leaders are focused on the people involved.

Finally, emotional intelligence is a relatively new leadership style scale (Goleman 1998). Since human beings' behavior is influenced by emotions, leaders can elevate emotions as a tool to improve leadership.

Tasks

1. Select three software development teams that you are familiar with. For each team describe its leadership style.
2. Discuss the characteristics of tightness (see Chapter 4, Time) with respect to leadership styles.

13.4.2 Case Study 13.1. The Agile Change Leader

One of the roles in a software team is that of the *methodologist,* who guides the team members how to follow the principles of their software development method and is responsible for the method's implementation (Dubinsky and Hazzan, 2006). This role definition stems from a team perspective, i.e., an insider to the team who constantly lives the methodology (see Chapter 2, Teamwork).

This case study broadens the discussion of this role, focusing on what happens in one specific organization in which several software projects and multiple teams participate in a transition process to agile software development (see Chapter 12, Change). In this case, the responsibilities and authorities associated with the role of methodologist are extended.

In general, an agile change leader, whom we call the *methodology change leader* (MCL), can be found in almost any transition to agile development. The MCL is an insider who knows well why change is required in his or her specific environment and finds it worthwhile to adopt the agile spirit either as a whole or only with regard to several of its practices. He or she is aware of the problems in the organization with respect to software development, can in most cases analyze them, and has reached the conclusion that the agile approach can solve them with some degree of success. The MCL can belong to the organization's management or to one of the software groups. In any case, the MCL is busy convincing the management to make the initial decision towards adopting the agile approach. This process can sometimes take months and even years.

In practice, the MCL is a kind of mediator between the different parties involved in the transition process. The MCL understands all sides involved and can put himself or herself in the shoes of either side.

The MCL role evolves gradually, since the MCL, as well as the other people involved in the process, learn while implementing the agile approach daily. An MCL should have enough patience and strength to lead a learning process that has its ups and downs, in which, on the one hand, everyone knows that the MCL is there for them, and on the other hand there are always some cynics who are just waiting to watch how he or she falls. In addition, the MCL establishes a network of team methodologists, who together serve as the backbone of the transition process and later on support the sustainability phase (see Chapter 2, Teamwork for scalability issues).

13.4.2.1 The Change Leader Model

In most cases, and as happened in this case study, when a large software development company that wishes to explore the introduction of agile software development into the organization, the MCL is placed in the center of this process. The MCL model presented in what follows highlights the centrality of this role in such transition processes. Based on it, we present several guidelines and specific practices that address the way in which the person holding this role should interact with the different players involved in the process.

As shown in Figure 13.1, the other entities in this model are the organization's management, the customer, the software team or teams, and consultants that most companies hire to help with the transition process (see Chapter 12, Change).

The MCL role, which is the heart of the model, plays a key role in the transition process, as is described in what follows:

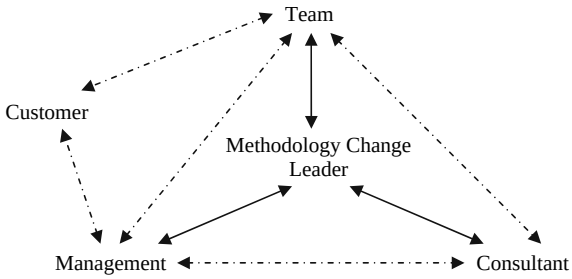


Figure 13.1 *Positioning the methodology change leader.*

- The MCL serves as a *knowledge center,* who is aware of all information related to the transition process, including plans, problems, and documents.
- The MCL serves as a *leadership center* in the transition process, who has the ability to lead the process and the people involved in directions he or she sees as being the best.
- The MCL serves as a *communications center*, who relays various kinds of messages among the different entities.

Each of the relationship lines in the figure represents a different kind of interaction and serves different purposes. The dashed lines represent interactions that are not made via the MCL. The solid lines represent interactions with the MCL. Note that since the MCL and the consultant are new to the organization and appear in the organization as a result of the transition process, the only interactions that existed prior to the transition were those depicted by the dashed triangle between the management, the customer, and the team. Still, since this model represents the transition, the relationship content of the triangle is changed by the transition to agile development. Another observation that emerges from using this model is that the MCL and the consultant are not related directly to the customer; they do, however, provide the management and the team with guidance and support on how to continue development according to the agile approach in general and how to mange the customer collaboration in particular.

In what follows, we describe the MCL's relationships in the model as they are manifested in the management-team-consultant triangle. In general, the MCL conveys the management's needs to the other people involved in the process, and returns feedback from the process to the management. In other words, the MCL establishes a communication and feedback channel among the entities, as follows.

Management ↔ MCL ↔ Consultant. Usually, there is no direct relationship between the consultant and the management other than via the MCL. When the MCL is absent and is replaced by someone else from the management, both sides will update the MCL on all that transpires. Specifically, in the Management ↔ MCL ↔ Consultant chain:

- Management ↔ MCL: The MCL works with the management in order to reach an understanding of the consultant's role and scope of authority, including the definition of the contract between the organization and the consultant. In general, the MCL is usually more concerned with the management's side.
- Consultant ↔ MCL: The consultant learns the measures used by the management to gauge the success of the process. The consultant reports to the MCL on all major events that the management should be aware of.

Team ↔ MCL ↔ Consultant. The main and most noticeable characteristic of this relationship is "presence;" the MCL is present in as many meetings as possible between the team and the consultant, especially during the methodology launching phase. Specifically, in the Team ↔ MCL ↔ Consultant chain:

- Team ↔ MCL: The MCL is involved in the selection of the best team to start the process in the organization, as well in the selection of any subsequent teams. The MCL encourages the team during the process, which is guided by the consultant, and delivers the message that the consultant's work is part of the management initiative. The MCL also listens to the teammates in order to understand their difficulties (some of which can be solved using the organization's internal resources).
- Consultant ↔ MCL: The MCL conveys feelings, problems, and requests that emerge in the team and works with the consultant to refine the way of work accordingly. In a sense, the MCL serves as the organization's "sensor." The consultant collaborates with the MCL in fostering the implementation of the agile methodology.

Team ↔ MCL ↔ Management. The Team ↔ Management relationship exists under all conditions and, in the transition process, receives additional attention in order to ensure success. Specifically:

- Team ↔ MCL: The MCL brings the management's vision to the team and shares with it the management's praises as well as its concerns. The team uses this interaction to deliver their fears and concerns. The MCL also searches for a team methodology leader (methodologist) in the different teams undergoing the transition process. Thus, the MCL builds a network of

methodology change leaders and receives constant feedback from the field on the agile implementation.

- Management ↔ MCL: The MCL delivers feedback from the team to the management and together with the management deals with special events concerning the team. The MCL is the management's representative in the Business Day of each iteration, and updates the management regarding the customer's level of satisfaction.

Tasks

1. Analyze the model relationships by the leadership styles presented in Table 13.2.
2. What characteristics should we seek for when hiring an MCL?
3. In your opinion, how can the agile approach assist the MCL in evaluating his or her role?

13.4.2.2 The Model Dynamics

The entities in the model are all people who are undergoing transition; thus, some level of dynamics can be expected. The MCL entity, in particular, is a dynamic entity that can move according to the subject matter. We now use Plotkin's notion of change (presented in Chapter 12, Change) and analyze two field scenarios to demonstrate the dynamic characteristic of the model.

Scenario 1. Several teammates talk with the MCL about their concerns with the way their role was changed. It is clear to the MCL that in order to improve the software development process in the team, these changes in role definitions are essential. The MCL tries to encourage them to see the positive sides of the change, but some still suggest they should leave the organization. The MCL decides that this situation requires the help of both management and consultant. Using this help, the MCL entity leans towards the team entity in order to solve the problem (see Figure 13.2).

Analyzing this scenario within the framework of coping with change, we can see the use of a *knowledge-gaining device* in the form of ongoing feedback. The MCL uses various pieces of advice received from the other entities involved in the process and focuses his or her efforts on solving *an isolated problem.*

Scenario 2. During the Business Day the customer expresses deep concern about the progress of a production release of a specific component of the developed product. The MCL feels that not enough is being done to address this issue

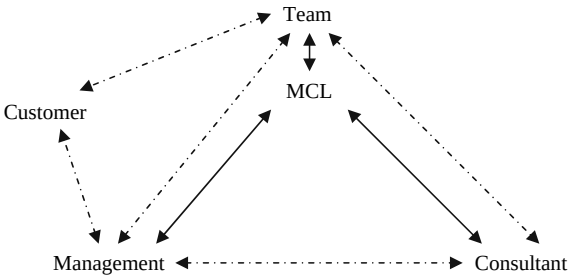


Figure 13.2 *Leaning towards the team.*

and suggests a plan to ensure that the production release is dealt with within the next few iterations. The MCL further stresses that this is the management's wish. The consultant suggests a measure to be used specifically to advance this component's progress, and the team's tracker adds it to the board for daily viewing. Figure 13.3 shows the MCL dynamics in this case.

Analyzing this scenario within the framework of coping with change, we can see that the MCL uses *time and scope reduction* in order to solve the problem within a few iterations. The MCL also adopts the *knowledge-gaining device* of measurements in order to control the process.

Tasks

1. Suggest two more scenarios that demonstrate the model dynamics and analyze them using Plotkin's framework of coping with change (presented in Chapter 12, Change).
2. Analyze the MCL role within the HOT—Human, Organizational, and Technological—analysis framework.

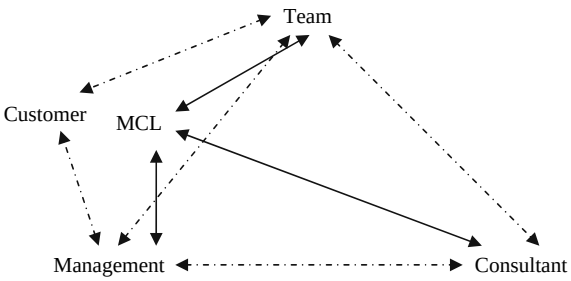


Figure 13.3 *Leaning towards the customer.*

13.5 Coaches

Coaching is often compared to sport coaching where the coach recruits players and develops their talents, implements strategies, establishes teamwork, and guides the players on how to improve their role performance (Topping 2002). The sport coach is measured *only* by the number of wins. In the case of a loss, the sport coach is considered the person most accountable for the failure.

In software development projects, the coach is the team leader. The agile approach relates to coaching as the "balance between being part of the team and having an independent perspective" (Beck and Andres 2004, 143). According to the perspective presented in Chapter 7, Learning, coaches improve their job performance during its actual accomplishment. The coach deals with communication problems and encourages the implementation of practices that enhance product and process quality (see Chapter 6, Quality). "A coach is responsible for the process as a whole, keeping the team working at a sustainable pace and continuing to improve ... a coach should encourage independence, not dependence. A good coach moves on a little before you think you're ready and leaves behind a team that finds itself firmly on a path to sustainable, profitable, stable, fast, fun software development" (Beck and Andres 2004, 143–144).

Tasks

1. Suggest three characteristics for a software team coach. For each characteristic, explain how it relates to the agile notions with which you are familiar.
2. Write a work procedure which describes the agile coach role. The work procedure should include the role goals, responsibilities, guidelines, and illustrative scenarios.
3. Referring to the work procedure above, suggest a way to evaluate the coach role. The evaluation description should include criteria for evaluation, ways to collect the required data, and methods of data analysis.
4. Analyze the coach role within the HOT—Human, Organizational, and Technological—analysis framework.

13.6 Leadership in Learning Environments

The atmosphere in this weekly meeting is characterized by positive stress and excitement towards the end of the semester. No special activity is planned with the academic coach. It is, however, an excellent opportunity for him or her to be

involved in the process, observe the group work as well as the individuals' contributions, and evaluate the product integration.

Students work to complete their development tasks and are busy with the integration process. They also complete their test-driven development exercise (see Case Study 6.4 in Chapter 6, Quality) and summarize their activities regarding their role.

In addition, the release presentation, scheduled for the next week, is discussed. See Chapter 14, Delivery and Cyclicality.

13.6.1 Teaching and Learning Principles

The inspiration for the nature of the software development approach is a teaching principle that is highly related to leadership in learning environments. In Chapter 1, Introduction to Agile Software Development, this principle was first introduced. Here we further connect it to leadership.

Teaching and Learning Principle 1: Inspire the Nature of the Software Development Approach

Leaders should inspire the software development approach instead of lecturing about it. This way the team members practice communication, reflection, and collaboration, leading themselves with the inspiration of high quality leadership.

13.6.2 Case Study 13.2. A Coaching Framework

This case study (based on Dubinsky and Hazzan 2003) presents a coaching framework, based on a reflective process (see Chapter 11, Feedback) conducted by a team of four coaches, who guided the development of agile software projects in academia during a full academic year. This coaching was performed in line with the academic coach description presented in Chapter 1, Introduction to Agile Software Development. For simplicity purposes, we use the term coach instead of academic coach.

Before the agile approach was introduced into the course, projects were developed with no emphasis on any specific software development method. Each coach had between five and seven groups of two or three students each. The students met with the academic staff at the beginning of the semester to receive their project requirements, and presented documentation and code on three specific occasions during the semester. During the rest of the time, the

students could approach the coaches during their office hours. The students usually did not use this means of communication. The coach was not involved in the students' work, since the students did not work near or with the coach when the actual planning, designing, or coding tasks were performed.

The group of coaches was trained using the agile approach before they started to coach. During the training sessions the team was introduced to agile basics, experienced several agile practices, and was trained to teach the method. Special focus was placed on the planning activity, since it provided a development framework from a time management perspective. The main issues of student team projects were discussed, including their structure, the coaches' roles, the students' roles, and the evaluation scheme (see Chapter 2, Teamwork). All decisions were made jointly by the entire coaching team.

Starting to work, coaches' meetings were conducted to rehearse agile practices. Specifically, guidance was provided to the coaches regarding which activities to conduct, how to overcome specific problems, what tasks to assign students the following week, and other such topics.

After a year of work, a reflective process was performed. The aim of this reflective process was to draw lessons from the accumulated experience, to be implemented in the following stages. Two almost identical questionnaires served as the basis for reflective interviews. The first questionnaire was filled out by each of the coaches prior to the first training; the second was completed towards the end of the year, when the coaches had had one year of experience. Both questionnaires referred to software development projects in general and did not address any specific software development method. In the questionnaires, the coaches were asked to describe the phases of a software development project, specifying the more important phases. In addition, they were requested to describe their role as coaches in the process of guiding students in software development; to rank the main activities performed during software development, as they perceived them; to specify the development process' main problems; and to suggest solutions to these problems.

Two consecutive interviews were performed with each coach. In the first interview, the coaches started by filling out the second questionnaire. Then they were asked to describe the agile implementation, the effect of the agile training (a year before), and the topics that they felt should be the focus of the next coaching training. In the second interview, each of the coaches was asked to reflect on the two questionnaires they had previously filled in and to describe how the use of agile software development influenced them personally. They were also asked to compare their teaching approach before and after this year. At the end of the second interview the coaches were requested to put in writing their reflections on the two interviews.

Additional details about the coaching training as well as about the reflective process, can be found in Dubinsky and Hazzan (2003).

13.6.2.1 A Coaching Framework

The data analysis yielded six categories that form a coaching framework, as shown in Table 13.3. The *Project* category emphasizes the management of resources needed for the project, such as a time schedule and various organizational aspects. The *Method* category addresses the method's practices and the tools used in the project. The *Development Team* category focuses on the development environment and on communication among team members. The *Customer* category brings the business aspect into the project and focuses on requirements and product acceptance. The *Feelings* category refers to the people involved in the project, from the viewpoint of their inner being. The last category, *Coaching Team,* emphasizes the support given to the coaching team in order to maintain continuous learning and receipt of feedback. Naturally, there are overlaps between these six categories; it seems, however, that each of these categories plays a significant role.

In what follows, we present the six categories, describing how each category was referred to by the coaches with respect to their coaching. In addition, we discuss the conceptual change related to each category.

Project. As it turns out, coaches must undergo some conceptual change in relation to the essence of coaching according to the agile approach. In our case, the coaches no longer perceived themselves as student project coaches, but more as project *leaders.* They held relatively long weekly meetings with the students, and as a result, were more involved in all phases of the project development. This in turn led to a more accurate project evaluation. In addition, the coaches were aware and appreciative of the resources they had in this case: a studio for each team, computers dedicated to the development of each specific project,

Table 13.3 A coaching framework (© [2003] IEEE)

Category	Description
Project	Resource management: time and organizational aspects
Method	Practices and tools used in the project
Development Team	Development environment and communication among team members
Customer	Customer requirements and product acceptance
Feelings	Inner being of people involved in the project
Coaching Team	Support given to the coaching team in order to maintain continuous learning and feedback

whiteboards, and other materials for ongoing activities. All of the above contributed to making them feel a higher commitment to their coaching role. Following are some of the coaches' self-expressions with respect to this category:

- I now have a much better feeling about how to lead a project. *Leading* a project by giving out the work and simply waiting for them [the students] to come to you with questions or results is one way. It's totally different, however, when you have a group, and you *lead* it through the project in one way or another. It gave me experience managing a project. Experience in different aspects: one is time scheduling, another is planning of the work, and another is the professional side of programming.
- This method makes me understand that first and foremost this work is with people.
- The frequency with which I see the students changed. It has both positive and negative sides. It is exhausting. It is tremendously exhausting. Especially when they are hard people.

As can be observed from the quotes above, the way projects are conducted using the agile approach demands more work from the coaches than they were accustomed to investing before. As it turns out, the coaches perceive this as a significant positive change. This may be a result of the fact that this framework provides solutions to some of the problems which coaches faced before the agile approach was introduced. At that time, coaches were not deeply involved in all aspects of the project and did not see the students actually work on the project. Within this coaching framework, the coaches were familiar with the project planning, design, coding, and testing, and thus they felt that they were part of the development process.

Method. The introduction of the agile method into academic environments requires the implementation of its practices in a way that fits this framework. Following are several examples of this. The small releases practice was implemented by structuring the course schedule around a release of three iterations (see Chapter 1, Introduction to Agile Software Development). The customer role was implemented, in cases in which no real customer was available, by having the coach as the main customer. In addition, one of the students was assigned to play the role of the customer throughout the duration of the course, especially during the planning sessions (see Chapter 2, Teamwork). The pair programming practice was implemented by installing approximately five computers in each project room and by encouraging the students to exchange pairs. When a metaphor was raised by the students in a natural manner, it was adopted by the coach as a means of increasing the students' understanding of the project's concepts (see Chapter 3, Customers and Users). The sustainable pace practice was adjusted to

the academic environment by the holding of two- to four-hour weekly meetings. Such meetings started with a stand-up meeting in order to align information. In addition, each student had to invest 10–15 hours every week working on his or her development tasks. Also, the use of the electronic forum in between meetings was encouraged to promote communication. Following are some of the coaches' self-expressions regarding several of the practices:

- I told them that no one works alone; working is done [only] in *pairs.* I told them of the case that happened last semester, when the coach [one of the students] left, and had held a great deal of code. They had to rescue what he had done.
- They worked *test-first* [. . .]. The group took the testing subject seriously, a real *test-first* and unit testing and then *automation* and *testing.*
- As for the *planning activity,* I really saw that it was important; they worked with the cards and wrote their contents into the computer.
- They [the students] built an *integration* machine. Anyone could have come and dealt with the code.

In light of the way coaching was done before the agile approach had been introduced, the change that took place as a result of its implementation in the course of a specific development method was significant. Specifically, based on the agile practices, the coaches constructed a framework for themselves and for their students, which made clear which activities to promote and which values to impart.

Development Team. With respect to the development team, we focus on the studio environment and the communications that it encourages (see Chapter 1, Introduction to Agile Software Development). As it turns out, the coaches had positive feelings towards this situation. They reported an increase in communications with the students and among the students themselves during the weekly meetings, and particularly during the planning sessions and code reviews. The coaches admitted that in this course setting, they talked to the students more and also became familiar with some of the students' personalities. Following are several of the coaches' self-expressions with respect to communications within the studio:

- Many problems are caused [in software development in general] due to a lack of communications. I saw, and I fairly agree, that many agile principles help to overcome the lack of communications. Communications between the customer and the development team, and also among the team members. Both are big problems [in other development environments].

- It was easy to improve communications since there are concepts [practices], guidelines provided by agile, they are fairly clear, and not so difficult to implement.
- The groups of three students that used to come to me before [before agile was introduced into the course], I barely remembered their names. I looked at what they did, never at their eyes, at who they [the students] were.
- [Coaching using agile is] much more active and much more alive.
- When working in groups of three, you threw this tool into the water, and waited for it to respond. You gave them a problem and you waited for them to come back in the middle or at the end of the semester. Here [in the agile environment], the entire process takes place right in front of your eyes.

As can be observed, communication increased significantly when the agile approach was used as the development method. Coaches began to appreciate the benefit of such communication, from both technical and social perspectives. Before the agile approach was introduced, when small groups of students worked on a project, the communication level between them and the coach was shallow and communication among the team members took place mostly at the end of the semester, towards the final presentation. In light of this, a conceptual change occurred in the coaches' understanding of the significance of communication.

Customer. Within the agile development framework, the customer's role became essential to all participants. Indeed, the coaches perceived the customer as playing a key role in their projects. Instead of talking about the course project, they started talking about the product. During the planning session, the students, on their part, elaborated on the stories provided by the customers or coaches. In projects with no real customers, during the iteration presentations, the students who had been assigned to be the customer first provided feedback on the current state of the product, trying to be as objective as possible. This was not always easy, especially during the planning session, when conflicts arose between the need to play the role of the customer and the awareness that issues raised by the customer would have to be dealt with and implemented. Following are some of the coaches' thoughts regarding the role of the customer:

- While reflecting [. . .] something new came to me based on the experience of the past year. It is an increased awareness of the customer, and of the human aspect of the development team in general.
- It was a real product that you could even "sell," a real on-the-shelf product that one could say, "Here, I have a product," so they got a real satisfaction from the fact that they actually did something.

- During the summer semester, we had a [real] customer. Moshe was the customer; he entered [the studio]. He really gave feedback. His sentences were in the background even when he was not participating. During the winter semester, we had a customer as well. [. . .] When he couldn't come, a student would replace him and was even in contact with him if needed. The issue of the customer is really important, and this semester it is missing. It is hard for the students to enter the customer's shoes since the scope is not clear. They are in conflict all the time.
- If a customer joins the project, it changes the coaching significantly.

In the past, the instructor wrote a detailed description of the requirements for each of the projects, so the students had no need to discuss requirements unless a question came up and was posed to the coach. With the introduction of the agile approach, a major change has taken place. Students were involved in the definition of the requirements together with the coach. The coaches reported that students were amazed at this process, and sometimes expressed their resistance by stating that in the previous format of the course requirements were given and were not subject to discussion. Being aware of the importance of experiencing the process of defining project requirements, the coaches had positive feelings about the students' involvement in deciding what requirements were to be part of a specific iteration.

Feelings. Personal feelings were expressed during the entire reflective process. Since all coaches addressed the human aspect of the agile approach, these expressions of feelings were natural. All coaches felt satisfaction with the new method and each of them emphasized his or her favorite practices. They also felt the satisfaction of the students in each of their studios. All the coaches expressed also the hard work involved in this kind of coaching. Following are several of the coaches' self-expressions of their feelings:

- There is a lot of personal satisfaction when a team arrives at the beginning [of the course] without having heard anything about the subject, or about me either, [. . .] and together we complete a project that starts at zero and achieves something. There is a lot of satisfaction when it succeeds. The benefit is the satisfaction.
- They have a real satisfaction from the fact that they really did something.
- I feel as if the project is my baby.

The coaches' high level of involvement led them to surface their feelings. Each coach met with his or her students every week, and after several weeks of working together got to know the students quite well. Thus, the coaches were highly involved, a fact that contributed to the project's progress and to the

communication among the people involved; however, this involvement should be monitored; otherwise, too great an involvement may lead the coaches to become subjective.

Coaching Team. With respect to the coaching team, we focused on the learning and feedback processes undergone by the coaches who participated in the training program (see Chapter 7, Learning, and Chapter 11, Feedback). In the first training session, the coaches became familiar with agile concepts in general and with several specific practices in particular (such as the planning session and pair programming). In addition, decisions about the course framework were taken. Following are some of the coaches' self-expressions about the first training session:

- The training offers a framework. It brings us to a certain level of knowledge that we weren't at before.
- The training taught me to pay attention to the people here. We are working with people, not with a subject [of a project]. My emphasis was always on the subject. The subject first. I must have been familiar with the subject as much as possible. This method [agile] makes me understand that this work is first and foremost with people.

Following are some of the coaches' self-expressions regarding their expectations for the next training session:

- What I am lacking now is the topic of time estimations. OK. They gave the time estimations, how we go and disassemble it.
- [I want to know more about] the second part of the planning activity, how exactly do you assign a function or a class to a student. I want to see an instructive example from beginning to end.
- I want to learn more about testing tools and how to build the integration machine.

As can be seen, the coaches would like the second training program to be dedicated to a more in-depth examination of the details of the development, as they are outlined by the agile approach. It seems that at this stage the new structure of the course has been accepted, and there is a need to fill it with more details.

Tasks

1. In what ways can you implement the coaching framework described in Case Study 13.2 for industrial environments? What elements can be adopted and what elements do not fit? Explain your answers.

2. Based on your experience, suggest additional aspects for the coaching framework described in Case Study 13.2.
3. Analyze the coaching framework presented in Case Study 13.2 within the HOT—Human, Organizational, and Technological—analysis framework.

13.7 Summary and Reflective Questions

1. Describe three events in software projects from the leadership perspective presented in this chapter.
2. Describe in detail your preferred leadership style. Describe your position—team member, team leader, project leader, etc.—and explain why this style suits you. Does this leadership style fit agile teams? Explain your perspective.
3. How can the contribution of your suggested leadership style be evaluated? Address its contribution both to the product and the process.
4. Explore the connections between the ideas presented in this chapter and Section 5: Management, of the Code of Ethics of Software Engineering, presented in Chapter 9, Trust.
5. Looking towards the release presentation next week, summarize your contribution to the project's management from the perspective of your personal role.
6. How did your team decide about the main messages to be delivered next week in the release presentation? What were the team's guidelines?

13.8 Summary

This chapter deals with leadership and coaching in software development environments in general and in agile software development environments in particular. Different leadership styles are described and a case study referring to the methodology change leader is presented. Also, the academic coach role is described, delving into the details of a specific case study that describes a coaching framework.

References

Augustine S, Payne B, Sencindiver F, Woodcock S (2005) Agile project management: steering from the edges. Commun ACM 48(12):85–89

Beck K, Andres C C (2004) Extreme programming explained: embrace change, 2nd ed. Addison-Wesley, Reading, MA
Bruce A, Langdon K (2000) Project management. Dorling Kindersley, New York
Ciulla JB (1998) Ethics: the heart of leadership. Praeger, Westport, Connecticut
Cockburn A, Highsmith J (2001) Agile software development: the people factor. IEEE Software 34(11):131–133
Cooper CL (2005) Leadership and management in the 21st century: business challenges of the future. Oxford University Press, New York
Drath WH (1998) Approaching the future of leadership development. In: McCauley CD, Moxley RS, Van Velsor E (eds) The center for creative leadership: handbook of leadership development. San Francisco, CA, Jossey-Bass, pp 403–432
Dubinsky Y, Hazzan O (2003) eXtreme programming as a framework for student-project coaching in computer science capstone courses. Proceedings of the IEEE international conference on software—science, technology & engineering. Herzelia, Israel, pp 53–59
Dubinsky Y, Hazzan O (2006) Using a role scheme to derive software project quality. J Syst Architect 52(11):693–699
Goleman D (1998) What makes a leader? Harvard Bus Rev 76(6):93
Hazzan O, Dubinsky Y (2005) cognitive and social perspectives of software development methods: the case of extreme programming. Proceedings of the 6th international conference on extreme programming and agile processes in software engineering, pp 74–81
Huff AS, Moeslein K (2005) An agenda for understanding individual leadership in corporate leadership systems. In: Cooper CL (ed) Leadership and management in the 21st century: business challenges of the future. Oxford University Press, New York, pp 248–270
Hughes B, Cotterell M (2002) Software project management, 3rd edition. McGraw-Hill, New York
Humphrey WS (2000) Introduction to the team software process. Addison-Wesley, Reading, MA
Mayrhauser VA (1990) Software engineering methods and management. Academic Press, New York
Nirenberg J (2002) Global leadership. Capstone Wiley, New York
Pulford K, Combelles KA, Shirlaw S (1996) A quantitative approach to software management—the ami handbook. Addison-Wesley, Reading, MA
Putnam LH, Myers W (1997) Industrial strength software—effective management using measurement. IEEE Computer Society Press, Silver Spring, MD
Sommerville I (2001) Software engineering, 6th ed. Addison-Wesley, Reading, MA
Topping PA (2002) Managerial leadership. McGraw-Hill, New York

14
Delivery and Cyclicality

Abstract

This chapter describes the cyclic nature of the agile software development process, which is composed of releases, each of which first, ends with the product delivery to the customer and a reflective process; and second, signals the beginning of the next release. We focus on the delivery of a product, to the development of which the entire release has been dedicated, describing what happens in agile software development at the end of the release—the delivery, as well as just before and just after it. Specifically, prior to the delivery, the customer examines the product and checks its fitness to his or her expectation; then, the product release is celebrated; finally, after the delivery, a reflective session is facilitated to explore the lessons learned during the release for the improvement of future developments. We also summarize in this chapter the teaching and learning principles presented throughout the book.

14.1 Overview

In Chapter 3, Customers and Users we saw how a release starts. One of the main characteristics of the beginning of a release is the role of the customer, who describes the project vision, the project main stories, and the guidelines according to which the development priorities will be set. The customer presents his or her perspective after the presentation of the previous release has been completed and

O. Hazzan, Y. Dubinsky, *Agile Software Engineering*,
DOI: 10.1007/978-1-84800-198-5_14,

a retrospective session between the two releases has been facilitated. This description reflects the cyclic nature of agile software development.

In this chapter we look at what happens between two releases, focusing on the end of the release—the period prior to the actual product delivery—the delivery itself, and the reflective process that takes place after the release delivery. This chapter also focuses on the customer, who gets a product that is going to influence his or her business value, after the entire release has been dedicated to the product's development.

In the section that focuses on learning environments, we summarize the teaching and learning principles presented in the various chapters of this book.

14.2 Objectives

- Readers will become familiar with the cyclic nature of agile software development processes.
- Readers will become familiar with delivery issues, such as the presentation of the release artefacts to the customer, the artefacts delivery, and the release acceptance tests.
- Readers will understand the atmosphere at agile software development environments between two releases.
- Readers will become familiar with the nature of metareflective processes—reflective processes on reflective processes.

14.3 Study Questions

1. What associations do you have with the two words appearing in the chapter title: delivery and cyclicality? What are their relations to the world of software development? What are their relations to *agile* software development?
2. In your opinion, what are the customer expectations towards the end of the release?
3. How would you suggest presenting the release product to the customer?
4. Review your personal reflective processes during the course learning. Describe the three most important lessons you learned from these reflective processes? Address lessons on the personal and team level.

14.4 Delivery

The period of time that takes place between two releases is highly important. The beginning of the release has been described in Chapter 3, Customers and Users. In this chapter we focus on the end of the release, which is a special event in the development cycle of software products. The customer gets the product he or she envisioned at the beginning of the release; the team members and the management get feedback with respect to how they accomplished and met the customer's needs. Indeed, at the end of each iteration the customer gives feedback to the team in the planning session; yet, these feedbacks are on a relatively low level of abstraction (see Chapter 8, Abstraction), focusing on the specific features developed in a given iteration. From a more global perspective of the release, the feedback addresses the product as a whole, referring to its global characteristics.

The period between two releases lasts about a week. It starts with several meetings with the customers in which the product features are reviewed and the team members make the final preparation for the delivery itself. Then one day is dedicated for the celebration of the delivery in which the product is presented to all the project stakeholders. Then a retrospective takes place. Finally, the planning of the next release starts (see Chapter 3, Customers and Users).

14.4.1 Towards the End of the Release

The period of time that takes place just before the end of the release is important, since the customer is going to get a product that will influence his or her business value. Therefore, several special activities should be carried out to accomplish the delivery successfully.

First, the team members work on the final product integration, making sure that all the tests are passed successfully. Several role-holders (see Chapter 2, Teamwork) are especially dominant in this period of time. The continuous integrator manages the integration process, while all the other teammates develop and run final acceptance (system) tests; the installer makes sure that the installation kit works properly and prepares a CD (or another tool) with installation instructions; the presenter organizes, plans, and prepares the presentation that summarizes the release.

The customer reviews the last iteration developments and checks global features of the product, such as cross release acceptance tests, response time, and GUI consistency. The professional relationships, shaped by the customer and the teammates during the release development, are expressed during this week, especially respect and trust (see Chapter 9, Trust).

This week is characterized by a high level of abstraction (see Chapter 8, Abstraction), examining the product as a whole. Such a perspective is based on the ongoing learning that has taken place during the development process.

Tasks

1. In your opinion, what is the most important characteristic of the "between the releases" period of time?
2. In your opinion, or based on your own experience, what feelings are mainly expressed during the "between the releases" periods of time?
3. Analyze the "between the releases" period of time within the HOT—Human, Organizational, and Technological—analysis framework.

14.4.2 Release Celebration

"Celebration! We deliver the software today! We are all here and we will start soon. Can you feel the excitement?" We invite you to feel this way when you read this section focusing on the celebration that takes place in agile software development environments at the end of the release. After the team and the customer have made the needed preparations (see the previous subsection), this celebration is carried out at the organization level.

In general, all the activities included in the iteration summary meeting (see Chapter 3, Customers and Users) are carried out also at the release celebration, but on a *higher level of abstraction* and with respect to a *wider scope.* The wider scope is expressed in two ways: First, the release scope is wider than the iteration scope; second, additional people, beyond the team, attend this celebration so it becomes an organizational/departmental celebration.

We call this section Release Celebration because we see it as an event that on the one hand celebrates the end of the release, and on the other hand marks the beginning of the next release. The end of the release is celebrated for several reasons.

First, the team has accomplished the tasks that had been allocated for each of the short iterations of the release. This has been achieved by the application of the various agile practices.

Second, at the end of the release the customer will want to thank the team for the efforts it has put into his or her product throughout the development process. Since the daily routine usually does not allow this, a specific event is a perfect opportunity for this purpose.

Third, a break is needed after the effort put out by the team in the tight development process that the agile approach encourages. A celebration is a perfect means for such a break.

Fourth, since after all, the celebration is a *professional celebration,* it can be used for some learning and reflective processes which will be applied in future development cycles.

And finally, the end of the release signals the beginning of the next release, which hopefully will introduce new challenges.

Tasks

1. What other reasons can you suggest for celebrating the end of the release?
2. What are your associations with the concept of celebration? How can they be applied to this celebration of the end of the release?

Among the many associations that come with parties, we would mention happiness, the gathering of many people familiar and less so, different decorations from the usual ones, special food, speakers. . . . Let's see how these associations can be applied to the release celebration.

Atmosphere. It is important to make sure that the celebration space includes some elements that indicate a celebration (we do not want to limit your imagination, so we do not mention any such elements). Also, light refreshments are served at the celebration. The atmosphere should allow everyone to be relaxed and without any duty to carry out during the celebration period. No one should feel under attack or criticized.

Presentation. While the presentation at the end of the iteration was at a relatively low level of abstraction (see Chapter 3, Customers and Users, and Chapter 8, Abstraction), examining the acceptance tests of all the customer stories included in the iteration, the presentation at the end of the release celebration is performed at a higher level of abstraction. Only cross-iteration, global features, and high level customer stories, which are intertwined and implemented in many stories, are presented with their acceptance tests. These stories reflect the nature of the product and how it can be used from a wider perspective.

Measures. As in the iteration summary meeting, measures are examined at the end of the release celebration also. These measures reflect the entire release and, from the measurement perspective, present a more global view of the release.

Tasks

1. Choose several stories from the different iterations of the project you are currently working on. Develop one story that would illustrate them all.

2. What measures should be presented in the release celebration? Are they the same as the measures presented at each iteration summary meeting? Should other measures be presented? Explain your opinion.
3. What additional information can measures at the release level deliver beyond the information provided by the measures taken at the iteration level?
4. Analyze the Release Celebration event within the HOT—Human, Organizational, and Technological—analysis framework.

14.4.3 Reflective Session Between Releases

So far in this book we have encountered several time slots which are not dedicated to explicit development activities. For example, at the end of the iteration, a Business Day takes place, part of which is devoted to a reflective process (see Chapter 3, Customers and Users); in Chapter 11, Reflection, we present a release retrospective that takes place between releases. Needless to say, though development is not carried out during these time slots, they are still very valuable for the development process. The importance of such breaks has been explained in Chapter 7, Learning, and Chapter 11, Reflection. Among other ideas, the contribution of these breaks to learning processes and to the improvement of future development processes is explained.

As presented in Chapter 11, Reflection, the end of the release is one of the cases in which we stop the development and dedicate time for learning purposes that will be used in the development of future releases. Since in Chapter 11, Reflection, we present a framework for a release retrospective, in the remainder of this chapter we present another kind of reflective process— a meta-reflective process—that can take place after one or several releases. This is, in fact, an expansion of the scope of the object of reflection. While in Chapter 7, Learning, the focus of the reflection was the iteration, and in Chapter 11, Reflection, reflective processes were conducted with respect mainly to project-related activities, the object of reflection on which we focus here is the reflective activities themselves—reflection and retrospection. In other words, the objects of thinking on which we reflect now are reflections and/or retrospections.

There are different ways to apply such processes. One option is to reflect on the reflective activity during its carrying out or just after it has been conducted; another way, which we illustrate here, is to reflect on a set of reflective sessions. Accordingly, we can facilitate either reflection on reflections, reflection on retrospectives, retrospectives on reflections, or retrospectives on retrospectives (see Table 14.1).

Table 14.1 Metareflective processes: reflective processes on reflective processes

	Reflection on	Retrospective on
Reflections	Carried out at the individual level with respect to personal reflections	Carried out at the team level with respect to personal reflections
Retrospectives	Carried out at the individual level with respect to team retrospectives	Carried out at the team level with respect to team retrospectives

Tasks

1. Discuss the main characteristics, advantages, and disadvantages of each of the cells of Table 14.1.
2. Among the four options presented in Table 14.1, in which would you prefer to participate. Why?
3. Apply with your team the four kinds of metareflective sessions described in Table 14.1. Analyze what happened in each case. What were the outcomes of each process? Which process yielded the most important outcomes? Why do you consider them the most important ones?
4. Going even further, we can facilitate a reflective process (either reflection or retrospective) on both reflection and retrospective sessions. Try this process. What are your conclusions?
5. Explain the idea behind the metareflective processes within Plotkin's perspective on change processes presented in Chapter 12, Change.

Case study 14.1 presents an example of reflection on retrospective processes.

14.4.3.1 Case Study 14.1. Metareflective Processes

This case study focuses on the metareflective processes that took place in the large-scale agile software project described in Case Study 5.1. In short, the project is a business-critical enterprise information system, considered to be highly complex and intended for a large and varied user population. The following description is based on Talby et al. (2006).

The data used for this case study were gathered from the personal reflections of the team members on the retrospective processes that took place at each of the iteration summary meetings. The data were gathered via written questionnaires several months after the retrospective sessions took place. The presented data are based on the first four releases (eight months) of the project.

The project's development team averaged 15 members during this period; this is an average, since the team experienced several personnel changes. According to

Whole Team practice, the development team included a mix of programmers, business analysts, testers, and managers. Note that although 15 people filled out the questionnaires used as the main data source, not all of them were in the team throughout the entire period. Still, the answers come from at least two people in each of the above roles.

As just mentioned, the team had a retrospective session every two weeks as part of the iteration summary meeting (see Chapter 3, Customers and Users), the intention of which was to discuss a specific problem in the development process, and make changes as necessary. The particular structure of this team's iteration summary meeting is presented in Table 14.2. Section 11.2 presents the team's guidelines for the retrospective session facilitated at the Business Day.

Figure 14.1 summarizes the team's answers to the following question: "Indicate the importance of each element of the iteration summary meeting." Possible answers were on a scale of 1 (unimportant) to 5 (very important). Team members were also given the option to explain their choices in writing.

The results indicate that, as a whole, team members considered the iteration summary meeting to be of high value—its average importance was 3.9. The most important element of the meeting according to this team was the customer's summary, with an average of 4.1. Note that in this case it was just a ten-minute element at the beginning of the meeting. It seems that the team members placed very high value on this direct form of feedback; as one member wrote: "It's hard to explain why, but it's good to know what he thinks."

The formal presentation of the system, as well as the reflection elements, received an average of 3.7. The respondents' explanations of their choices were usually in line with the intended goals of these elements (see Chapter 3, Customers and Users). The review of metrics element received an average importance of 3.0, and the written comments supported the impression that team members were divided in their opinion regarding its importance. All managers and team leaders viewed it as highly important, while some of the novice team members wrote that "it is mainly of interest to managers." In this specific case, these results reflect an inherent difficulty in balancing some programmers' general dislike of "management."

Table 14.2 Example of the structure of an iteration summary meeting (© [2006] IEEE)

Schedule	Activity
9:00–9:10	Customer's summary of the iteration
9:10–9:25	Formal presentation of the system
9:25–9:50	Review of iteration's metrics
9:50–10:45	Retrospective

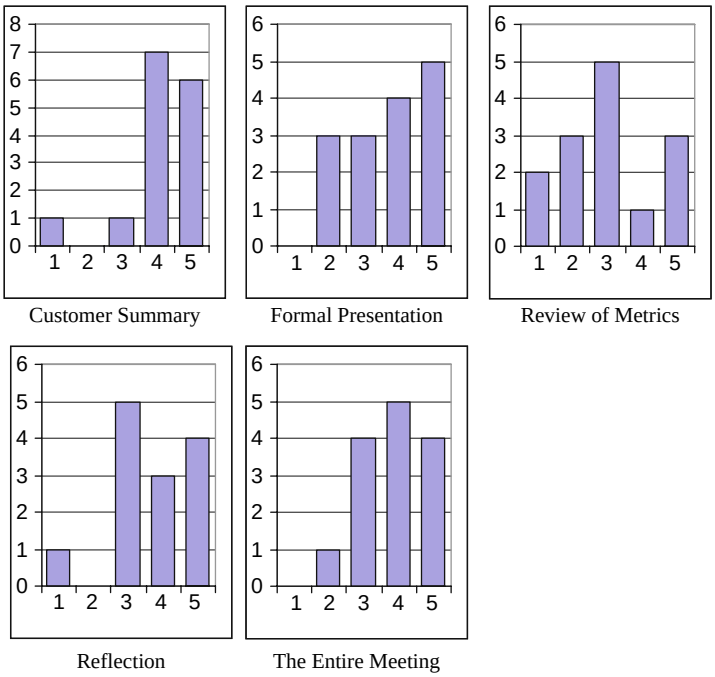


Figure 14.1 *Perceived importance of elements of the iteration summary meeting (© [2006] IEEE).*

When asked the open-ended question: "Which elements of the meeting should be modified (extended, reduced, or cancelled)?" the most popular answer was that reflections could be *extended* when their subject was very important. There were no suggestions to cancel any element, and (this was asked in a separate question) no offers of any new elements.

Figure 14.2 summarizes the team members' reflections about the goals of the reflection process. They were asked about their agreement with given statements, and possible answers were: strongly disagree (SD), disagree (D), indifferent (I), agree (A), or strongly agree (SA).

As Figure 14.2 shows, the team members generally assessed the reflective sessions carried out in this project to have achieved their goals as defined for this project.

Task

According to the data presented in Figure 14.2, can you predict the atmosphere in this team's retrospective sessions?

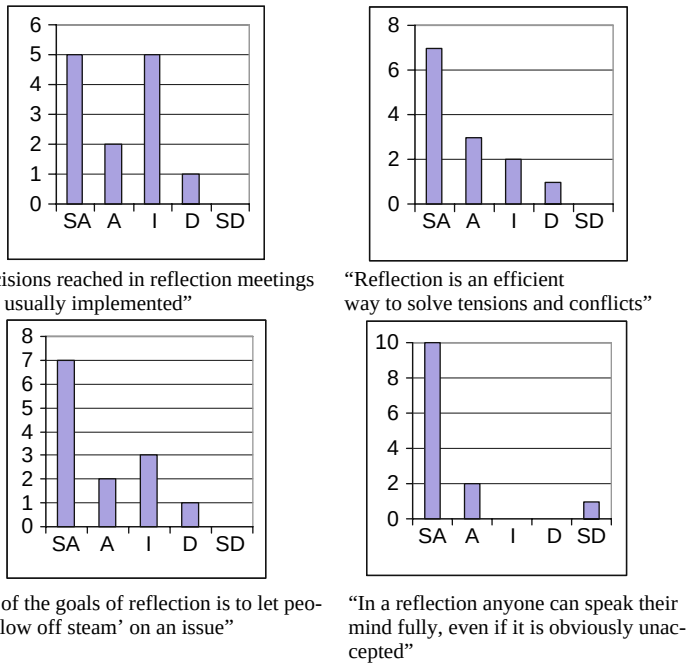


Figure 14.2 *Reflections on the goals of the retrospective meetings*(© [2006] IEEE).

Let us focus on the responses indicating that these reflective sessions served as a tool to vent negative feelings on an issue (average 4.2). The main cause for this was that in these sessions, team members were able to express their true opinions and feelings about the debated subject, even if it was obviously unaccepted by the rest of the team. This statement had an average agreement of 4.5, with only one who disagreed, writing that personal insults should not be allowed. Blunt and sarcastic statements were sometimes a natural part of retrospectives, but the results suggest that the positive effects of the retrospective sessions, of creating an open and honest atmosphere, outweighed their negative effects.

The questionnaire filled out by team members included also several questions intended to measure specific aspects of the reflective session. We focus on four aspects, summarized in Figure 14.3.

First, team members assessed the subjects of reflective meetings to be relevant to their ongoing work (an average of 4.1, no one disagreeing). This is important, since in this team the subject was chosen in advance by the moderator (who was usually the team leader). This had the advantage of not having to spend time deciding on the subject during the reflective session itself, thus lowering the overall time it required. The risk of selecting "wrong" subjects did not seem to

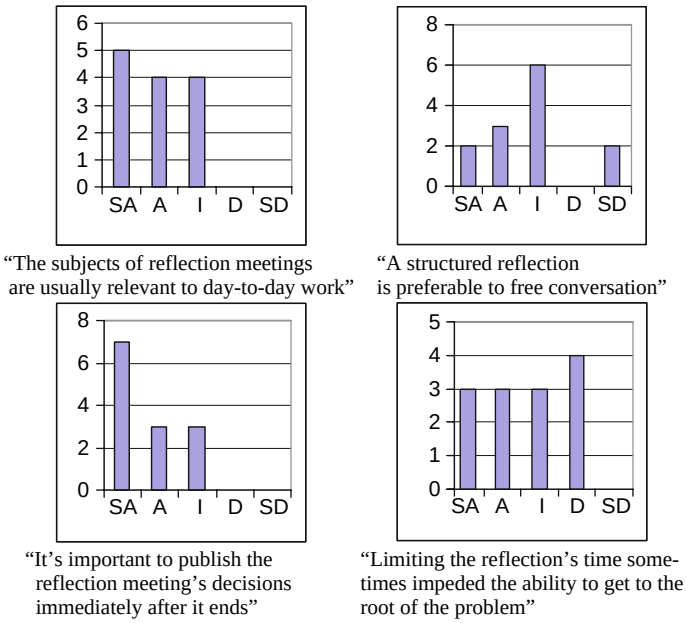


Figure 14.3 *Reflections on the technique of the reflective sessions (© [2006] IEEE).*

materialize in this project, probably because the moderator was the team leader, who was a member of the development team and was thus intimately familiar with its day-to-day problems.

Second, only five team members agreed that structured retrospectives were preferable to unstructured ones (an average of 3.2). Only four reflective meetings were structured throughout the period examined. It may be the case that the right problem-solving technique was not introduced to this team. In any case, open discussions were sufficient to achieve the bottom-line positive results for the team.

Third, the publication of a written summary of each reflective meeting, even if done in an informal forum such as email, was perceived by the team as highly important (a 4.3 average, no one disagreeing). In their comments, people explained that this was important mainly to prevent arguments on whether something was agreed upon and in what exact way, particularly as the project's history became longer and the risk of forgetfulness rose. An additional benefit was enabling newcomers to the team to catch up quickly with the team process and general insights that the team applied.

Fourth, the team had mixed opinions as to whether the strict limit on the reflective session's time frame impeded its effectiveness. The average agreement on this statement was 3.4, and the written comments on the issue were mostly of

two kinds: either that more time was required only in several difficult retrospectives; or that in some reflective sessions much more time could have been spent, but would not have led to any improved results. It may be that when trying to balance a fruitful discussion, a minimal overhead, and a thorough investigation of the given problem, maintaining a strict one-hour limit works against those goals.

Having discussed the technique and the goals of the reflective session, we turn now to the most important question: *Is it effective*?—the answers to which are summarized in Figure 14.4. Briefly, all results were very positive.

Team members highly agreed that raising a problem in a reflective meeting is better than having a decision made by the team leaders alone (a 4.1 average, no one disagreeing), and that they would be glad if were used in their next team (a 4.2 average, no one disagreeing). Nor did anyone agree with the statement, "I don't understand at all the purpose of reflections" (a 1.5 average). It seems that people are much more satisfied when they take part in the decision making process, and when their opinions are seriously heard.

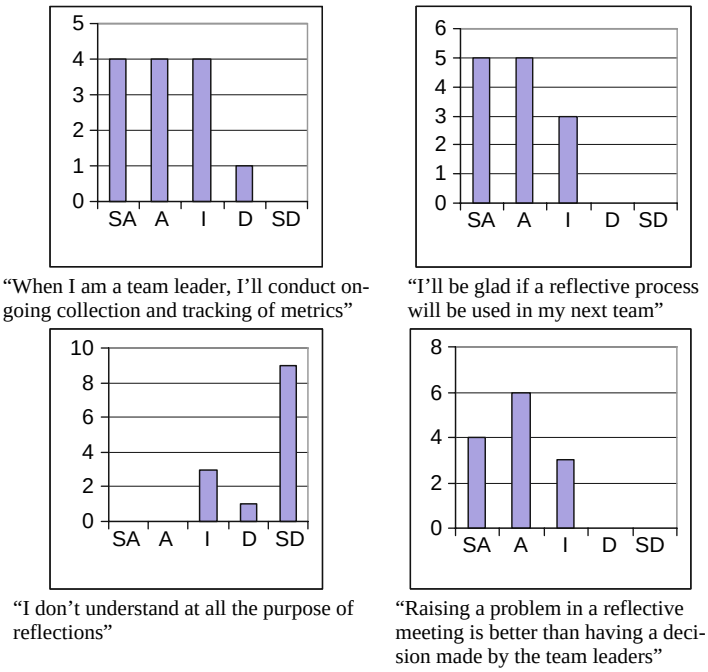


Figure 14.4 *Reflections on the perceived effectiveness of metrics and reflections (© [2006] IEEE).*

This case study analyses the reflective practice of an agile team in a large-scale, long-term software project in an industry setting. It is perceived as effective in stabilizing a new agile project, fostering continuous improvement, and resolving team conflicts.

Task

What would your answers to the questions presented in Case Study 14.1 be if your team had conducted a reflective process on the iteration basis and a metareflective process as presented in this case study?

14.5 Cyclicality

When the product of the release is delivered, the customer gets a version of the product that can be used; this, however, does not signal the end of the development. In most cases, additional releases are needed to complete a product. That is, a release ends but the development continues. In other words, the development process is cyclic—it is composed of several releases, each of them including several iterations.

Cyclicality is reflected not only by the fact that each release has several iterations, but also by the fact that the software life cycle is composed of several releases. In practice, activities that are performed between iterations are carried out on a larger scale between releases. Specifically, the cyclicality is expressed by iteratively performing the following activities: planning according to customer stories and priorities; development, including exhaustive testing; presentation and feedback; a reflective process. This kind of cyclicality is analogous to the iterative life cycle of other cases. In Dubinsky and Hazzan (in press) we elaborate on the analogy of this idea to teaching agile software development.

The difference between iterations and releases is expressed, in the context of this chapter, in the delivery process. While at the end of each iteration, the product is potentially deliverable; at the end of the release, the product, in most cases, is actually delivered. Still, there are occasions in which the splitting into releases is artificial and is made simply in order to keep the development in small releases.

It is important to realize that though each release and iteration has a similar structure, the developed product keeps changing, evolving, with the goal of satisfying the customer's expectations and improving the customer's business value.

Tasks

1. Identify additional cyclic elements in agile software development.
2. Are you familiar with other cyclic phenomena in your life? In what way is their cyclicality expressed?
3. Find connections between the cyclic nature of agile software development and Plotkin's framework for coping with change, presented in Chapter 12, Change.
4. Analyze the cyclic nature of agile software development within the HOT—Human, Organizational, and Technological—analysis framework.

14.6 Delivery and Cyclicality in Learning Environments

This chapter brings us to the end of the book and the course learning. This is definitely an appropriate reason for celebration. We close a specific period of time which was dedicated for a specific purpose—the course learning—and start a new period during which you, the readers, will use what you have gained from the course learning.

Similar ideas to the ones presented in the first part of this chapter are applied to the celebration of the end of the course learning. As in the real development environment, the last week of the semester is dedicated to meetings with the customers (either proxy or real), integration activities, preparation for the release presentation, and reflective session(s) that take place after the release celebration.

In addition to the description of how the end of the release can take place in learning environments, we summarize in this section the teaching and learning principles presented in the various chapters of the book.

Task

In your opinion, what do the concepts *delivery* and *cyclicality* mean in the learning of agile software development?

14.6.1 The Delivery in the Studio

This studio meeting celebrates the end of the release development in the course component in which the students developed a software product (see Chapter 1,

Introduction to Agile Software Development). During the last week, on the team level, the students made the needed preparations for this meeting. In the meeting, all students from all teams gather for a real celebration with light refreshments. Each team reports on its main achievements with respect to its project, and then a tour is conducted among the different studios to enable the students to visit and examine the projects developed by the other teams.

After the tour, each group meets with its customer(s) and academic coach for the presentation of the first release that was developed in three iterations during the fourteen weeks of the semester. The academic coach facilitates a feedback and summary session of the project.

Table 14.3 presents an example for a timetable of such a release celebration in academia for a course with four teams.

Here is a short explanation of each component.

Project presentations. The lecturer starts with a brief general explanation of the structure of the meeting and its purpose. Then each team presents its project in front of all the teams for about ten minutes, with five minutes for quick questions from members of the other teams. These presentations both contribute to the students' learning by requiring them to encapsulate the essence of their project in several minutes, and foster information sharing.

The presentations are at a high level of abstraction. In most cases, the presentation of each project includes an overall project description, the main requirements, what has and has not been achieved, the work process (including the gradual project evolution over the three iterations), the general design measures, the problems encountered and how they were solved, what has been learned during the development process with respect to the project subject and software project development in general, possible future development, and the project illustration.

Open tour in projects' studios. The tour of the various studios allows the students to see in more depth what has been developed by the other teams and to ask additional questions according to their interests. This component of the end of the release celebration widens the students' perspective on the development process with respect to what has been developed and learned in the other teams.

Presentations and discussion in the team studio. The last part of the celebration takes place in the studio, each team with its customer(s) and academic coach.

Table 14.3 Example of a celebration timetable at the end of the semester

Schedule	Activity
14:30–15:30	Project presentations (15 minutes for each team)
15:30–16:00	Open tour in projects' studios
16:00–17:00	Presentation and discussion in the team studio, including the presentation to the customer and a feedback and summary session

At the beginning of this part, the product is presented to the customer. Unlike what is done in industry (see Section 14.4), the presentation to the customer in the academic setting is done by reviewing *all* customer stories. This is because in the academic setting, this time marks the project's end; most probably the team will not meet the customer again, as they would in the industrial case, where project development proceeds to the next release and the interaction with the customer continues. This presentation format highlights the importance of acceptance tests. In general, any problems encountered by the team are raised in this presentation; specifically, if the team did not develop all the allocated stories, the reasons for this are explained. When a specific question is raised by the customer, the relevant role holder answers it.

After the product is presented to the customer, the team and the academic coach facilitate their last feedback meeting, which includes general comments from the coach and the students' reflective processes. It is advisable not to make this conversation too structured and to let the students navigate it according to their needs. At this stage of the semester they are familiar with each other and are capable of doing this. When appropriate, the academic coach can direct the discussion, focusing on questions such as what could have been done in the second release (had it been developed), and what considerations should be considered in order to continue the project development.

14.6.2 Teaching and Learning Principles

This section summarizes the eleven teaching and learning principles that guide the course pedagogy, presented in the various chapters of the book (see Table 14.4). These principles can serve as pedagogical guidelines for the teaching

Table 14.4 Teaching and learning principles

Principle	Description
1	Inspire the nature of the software development approach
2	Let the learners experience the software development approach
3	Explain while doing
4	Elicit reflection on experience
5	Elicit communication
6	Establish diverse teams
7	Assign roles to team members
8	Manage time
9	Be aware of abstraction levels
10	Use metaphors or "other worlds" concepts
11	Emphasize the software development approach in the context of the world of software development

of any software development approach and can be applied in both academic and industrial settings. For additional details, see Hazzan and Dubinsky (2003, 2006, 2007).

As can be observed, the numbers of the principles as presented in the various chapters are not consecutive. Indeed, while in the chapters we presented them in the appropriate context, within one teaching and learning framework the following order reflects a more coherent picture.

14.7 Summary and Reflective Questions

1. In your opinion, if the software product developed in the studio continues to be developed after the semester ends, what directions will its development take? Based on what information did you base your conclusions?
2. What main lessons did you learn during the project presentations?
3. How are the lessons learned connected to the main ideas learned in the course? Address the three aspects of the HOT—Human, Organizational, and Technological—analysis framework.
4. Analyze the ideas presented in this chapter within the HOT—Human, Organizational, and Technological—analysis framework.
5. What main ideas did you learn in the course?
6. Define "agile software engineering."
7. Present the main ideas of this book within the framework of one main idea.
8. What further issues would you like to learn about with regard to agile software development? How would you intend to learn them? Ask yourself questions such as: What do I take from the course, and how do I continue its learning? Can you use the metaphor of the agile process for the analysis of your learning process?

14.8 Summary

This chapter ends the first release of the course learning and, for the interested learners, starts the second release. We have presented in this chapter the period of time that takes place between two releases, highlighting the ideas of delivery and cyclicality to emphasize the continuous nature of agile software development with delivery peaks.

References

Dubinsky Y, Hazzan O (in press) Action research in software engineering: using a 3D analogy to analyze the implementation of agile software development in software teamwork. Computer Software Engineering Research, Frank Columbus (ed). Nova Science Publishers

Hazzan O, Dubinsky Y (2003) Teaching a software development methodology: the case of extreme programming. The proceedings of the 16th international conference on software engineering education and training. Madrid, Spain, pp 176–184

Hazzan O, Dubinsky Y (2006) Teaching framework for software development methods: poster presented at the ICSE educator's track. Proceedings of ICSE (International Conference of Software Engineering), Shanghai, China, pp 703–706

Hazzan O, Dubinsky Y (2007) Teaching agile software development quality assurance. In: Stamelos I, Sfetsos P (eds) Agile software development quality assurance book. Idea Group Inc., pp 171–184

Talby D, Hazzan O, Dubinsky Y, Keren A (2006) Reflections on reflection in agile software development. Proceedings of the agile conference, Minneapolis, Minnesota, USA, pp 100–110

Epilogue

This book focuses on the agile approach to software engineering.

We end this book by presenting contemporary issues as food for thought: To what extent will the agile approach be accepted as *the* approach for the development of software products? What will be the needs of the software industry in the future? Will agile software development fit these needs? Will a different approach for software engineering processes be needed? If so, what will be its main characteristics? How will it be accepted by the software industry? Will its assimilation process be similar to the assimilation process of agile software development? Will the agile approach be adopted for non-software projects?

No matter what the answers to the above questions are, we believe that the basic ideas of agile software development and the HOT—Human, Organizational, and Technological—analysis framework for software development methods remain valid.

Index