

Chapter 6

Enabling Lean Software Development

*Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza.*

*(Consider ye the seed from which ye sprang;
Ye were not made to live like unto brutes,
But for pursuit of virtue and of knowledge.)*

Dante Alighieri, *Divina Commedia*, Inferno, Canto 26, 118–120

*Lazily, Uli went back to his office. He called Euril, Perim, Sinon, and Elp. “Knowledge, value, improvement—this is what they asked me to clarify. . . Any suggestion guys?” None spoke; they were all still a bit shocked of all what happened the day before. “Well, I had a meeting with our top managers and actually there were not so excited of keeping the project running. It appears that for some of them not altering the consolidated way of doing business is more valuable than keeping a project and trying something new. Can you believe this?” “Indeed, said Perim, indeed. The good-old-way does not jeopardize any position while trying something new may alter the current status of businesses. But you should also consider the point they raised. I do think that there is something to think. Yesterday I asked something similar and you answered with a typical consulting term. . . what was it? Situational awareness—the usual consultant b***s**t!”*

Uli stood, joined his hands “Ladies, gentlemen” he started “can you consider why you started to work in software? What were your aims, your desires? All of you knew at the time that this was the land of unknown, the land of discovery, the land of opportunity. You selected software engineering because you aimed to increase your knowledge and the knowledge of the human being, still being able to do something concrete; you wanted to pursue your virtues. If you wanted to stay in a more comfortable, secure, quite discipline, you would have built houses and not software systems. If you wanted to be creative but not attached to the reality, you would have studied management. But now, you are a software engineering—someone who combines quest for the unknown and the ability to build solid elements, someone who put knowledge, and also value and improvement at the top of his priorities.”

None said a word. Uli sat. Everyone was silent and it was a very loud silence. The walls were speaking: two posters dominated the scene. On one side a big poster contained an artistic painting of stars in the Austral hemisphere on the 9th of April. It was a bit surreal but beautiful. In the spare time Uli used to joke with his senior architects telling them that he would have liked to be the first western man crossing the ocean to see such scene, or that he should have rented the space shuttle to take them all for a space tour as a prize for completing the project on time. On the other side there was the picture of an ancient Greek ship sinking during a horrendous nightly hurricane, with a mountain in the background and under a pale light of the moon; this picture had always been there, Uli disliked it but never took it away, as, he said, reminded always the risk of a failure.

After this heavy silence and after staring at the two pictures Perim said: “I want a ticket for the space shuttle!” and left the room. All the other also left.

At that point Uli knew what to do.

6.1 Introduction

The concept of Lean Software Development refers to the several attempts to transfer and adapt the principle of Lean Management into Software Engineering.

Remember, Lean Management was conceived in Japan by Taiichi Ono [21] for the automobile industry. Posting it to the software industry is not straightforward.

At the time of writing this book, a detailed methodology that fully applies the principles of Lean Management to Software Engineering does not exist yet. We have the impression that existing approaches do as if writing software would be similar to producing a car and ignore that software is invisible. A comprehensive measurement approach is needed that is aligned with the organizational goals as evidenced by the founders of Lean.

Current approaches emphasize different values of Lean but neglect the need of instilling a measurement and an experience management culture to overcome the invisibility of software and to improve its development constantly.

6.2 Existing Proposals to Create “Lean Software Development”

Practitioners and academics are exploring the terrain to adopt Lean ideas within software development. Pioneers in this exploration are Mary and Tom Poppendieck. In their book “Implementing Lean Software Development From Concept to Cash” [23], they characterize Lean software development with the following seven principles:

1. Eliminate waste;
2. Build quality—we used the terms “autonomation” and “standardization”;
3. Create knowledge;
4. Defer commitment—we used the term “just-in-time”;
5. Deliver fast—get frequent feedback from the customer and increase learning through frequent deployments;
6. Respect people—we used the term “worker involvement”;
7. Optimize the whole—we used the term “constant improvement.”

Furthermore, to respect people means that the knowledge people accumulate during their work is acknowledged and that they are given the possibility to change the working processes. According to them, this has consequences on different levels:

- **Entrepreneurial Leadership:** the leader promotes committed and thinking people and concentrates their efforts on creating a product that provides maximum value to the customer.
- **Expert Technical Workforce:** the company makes sure that the technical expertise is nurtured and that teams have the expertise needed to accomplish their goals.
- **Responsibility-Based Planning and Control:** teams are organized using “Management by Objectives” (see Chap. 4) and people are trusted to self-organize to achieve their goals.

Curt Hibbs and his colleagues have developed a different proposal to adapt the principle of Lean Management to Software Engineering. Their approach is more oriented to the code. In their book “The Art of Lean Software Development, A Practical and Incremental Approach,” [13] propose the following practices:

1. Source Code Management and Scripted Builds;
2. Automated Testing;
3. Continuous Integration;
4. Less Code;
5. Short Iterations; and
6. Customer Participation.

These principles are an implementation of the concept of autonomation to coding.

The use of source code management and scripted builds together with automated testing is one way to instantiate autonomation for software engineering.

Automated, scripted builds and automated testing automate coding because they detect if the produced source code does not conform to the expectations and stop the production process. Therefore, they contribute to avoid committing defective code to the production code.

Continuous integration is also a form of automation. By continuously integrating the different parts of code, all the problems related to integrating different parts of the software system become immediately evident and the production process does not proceed forward until the issues that break the integrations are solved.

Short iterations and customer participation enable the team to obtain frequent feedback and to improve the understanding of what creates value for the customer.

In summary, the proposal of Mary and Tom Poppendieck aims to “leanify” the overall process while Hibbs and his colleagues specify how coding can be made Lean.

Alan Shalloway et al. [25] take a more comprehensive perspective and propose a “Lean-Agile software development.” Their approach is more generic than the two previously presented and organizes the transition of Lean Thinking to Lean Software Development into the following layered model (see Fig. 6.1):

1. **Foundational Thinking.** The underlying belief system of Lean Thinking, based on the work of Deming.
2. **Perspective and Principles.** The Perspective is the choice of what is considered important to observe in the process. The Principles are the rules of behavior that adhere to the Foundational Thinking and are taken from the work of Mary and Tom Poppendieck.
3. **Attitudes.** The choice of what is considered important and what is not.
4. **Knowledge.** “Know-how” based on experience or, in other words, “lessons learned.”
5. **Practices.** Recommendations on what to do, based on the knowledge acquired.

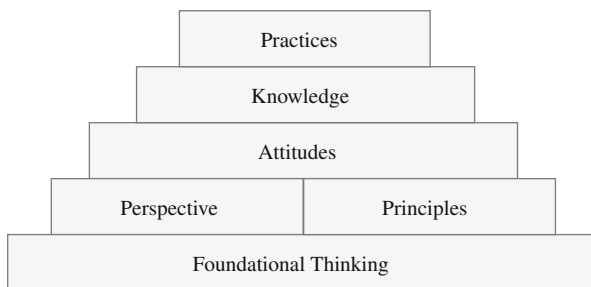


Fig. 6.1 The layered structure of Lean-Agile software development [25]

What is continuous integration?

Martin Fowler and Kent Beck were the first to write about continuous integration. We use ten practices proposed by Fowler [10] to explain this concept:

(continued)

1. **Maintain a single source repository:** maintain all resources of one software project in one place;
2. **Automate the build:** automate all steps to transform source code into a running system.
3. **Make your build self-testing:** include automated tests in the build process and execute them after building a new release.
4. **Everyone commits to the mainline every day:** the more often everybody commits to the mainline, the lower the effort of resolving conflicting changes by different developers becomes.
5. **Every commit should build the mainline on an integration machine:** because of different reasons (e.g., undisciplined developers, environmental differences between the developer machine and the integration machine, etc.), tests can still fail on the integration machine. Therefore, every commit should start an automatic build and test on the integration machine.
6. **Keep the build fast:** the faster the build, the faster the feedback that is given to the developer, and the lower is the risk that other developers are making their modifications based on the defective code, increasing the caused damage.
7. **Test in a clone of the production environment:** test your build in an environment that is similar to the production environment.
8. **Make it easy for anyone to get the latest executable:** put the latest executable on a well-known place to allow demonstrations and exploratory testing, find out about changes, etc.
9. **Everyone can see what’s happening:** communicate to everybody the state of the build.
10. **Automate deployment:** to test the developed code in multiple environments, it is important to automate the necessary deployment steps.

Table 6.1 contains some of the tools currently available to support the different phases of continuous integration.

Table 6.1 Tools to support continuous integration

Tool	Useful for step
Subversion [2], GIT [12]	1
Make [11], Apache Ant [1]	2
Unit testing frameworks (known as xUnit frameworks) such as Junit for Java [16] or CppUnit for C++ [7], GUI testing frameworks such as Sikuli [26]	3
CruiseControl [8], Jenkins [15]	5
VMWare [31], VirtualBox [22]	7, 10

The three examples presented above (the proposal of Mary and Tom Poppendieck and of Curt Hibbs and his colleagues and the approach of Alan Shalloway et al.) show that Lean Thinking can be translated in different ways into software engineering. However, all these three approaches lack an essential component of Lean Management, its concrete use of real measurements supporting the process [19]. They are more faith-based, while Lean advocates a constant and concrete analysis of the process to produce value and eliminate waste.

Concretely, our approach is to develop a Lean software development process that avoids the three issues we identified in the previous chapter:

1. the problem of communicating the goals and methods of Agile methods to stakeholders, which generates skepticism since Agile methods seem to ignore “well-known” best practices;
2. the guru approach that has dominated the way Agile ideas became known among practitioners; and
3. Agile extremists that promote the dark side of Agile.

Now we describe how we want to tackle these issues.

6.3 Share a Common Vision

Lean Thinking advocates new, unconventional methods for producing goods. It is essential that these methods can be explained solidly to our customers. They should not think that we are “original.” They should understand such methods and, at least, understand that they are grounded in solid theories. Otherwise, we would not have customers, or, worse, we might get customers who want to adopt our proposals simply because they are cool, and when their coolness will go away, we will not have anymore a job. It is interesting to note that several Agile projects had this fate, despite being successful.

Therefore, we need to communicate to our customers how we work, what we do, what outcomes we expect from it, and which support we need from them.

Agile methods heavily rely on the collaboration with the customers. Extreme programming, for example, has a practice called “customer on-site,” requiring customers to sit with the project team throughout the project and to supply the essential knowledge of the applicative domain whenever needed and to help the team to stay focused on the common goal.

However, customers have their own priority. In most cases, their ideal relationship with the developers is that they communicate shortly their desires, and, after a certain amount of time (the sooner, the better), they get what they dreamed at. Van Deursen [30] has identified three major causes why it is hard to have customers on-site.

Actually, it turns out to be difficult to convince the customer that it is worth to collaborate personally and continuously [30]:

- customers have to do their regular work and be on-site, which is not always possible;
- the customer usually wants to buy a “whole solution,” and not to run a customization project requiring his involvement; and
- the best customers from a programmer’s perspective are also often best in other aspects, which makes them busy, and it is unlikely to allocate to the project all the required time.

Some customers expect software development to be like building a house; they want the “whole solution,” the “turn-key project.” They want to get the solution in a ready-to-use condition. Such customers think: “Why do you ask me? You are the expert, you should know. Why am I paying you?”

Having the customer on-site, we are only halfway through: establishing a fruitful communication with the customer is also challenging. Some reasons for this are [30]:

- Technologists and end users have a high “semantic gap,” which makes communication complicated. Both sides base their communication on assumptions. If some information is based on an assumption that the other side does not know about, this information might be not interpreted as intended by the speaker. Making these gaps explicit, i.e., talking about the hidden underlying assumptions, is perceived as an annoying, boring activity.
- Neither developers nor customers consider talking to each other a useful task, but rather a waste of time.
- End users may resist changes in their way of working, making it very hard to involve them in a constructive way in the customization of the product.
- Developers might be against an on-site customer. Beck and Andres [4] call it the “sausage factory” effect when the developers think: “if the customers knew how messed up software development was, they would never trust us.”

From a financial point of view, a trade-off exists between having a customer on or off site. Let us analyze the trade-off looking at a client organization “BusyClient” that hires the software organization “AgileCoders” to produce a tool called “SuperTool.” BusyClient has to decide if its best sales representative, Mr. Seller, should act as a customer on-site.

If Mr. Seller works as an on-site customer at AgileCoders, he is not working as a sales representative, but as a requirement analyst. He can work a bit while sitting at AgileCoders, but he cannot leave to visit customers. This causes considerable costs for BusyClient and increases the total development costs of SuperTool.

The alternative is that Mr. Seller helps only off-site, which makes it harder for AgileCoders to obtain a clear understanding of the requirements since Mr. Seller is sometimes busy and not reachable when he is talking with customers.

AgileCoders might waste time because of misunderstandings, wrong assumptions because Mr. Seller is not available, and so on. This might then delay the shipment of SuperTool, which increases again the costs for BusyClient.

Additionally, because the shipment of SuperTool is delayed, AgileCoders is also facing higher costs compared to a scenario in which Mr. Seller acts as a customer on-site. AgileCoders, which wants to survive on the long run, has to charge BusyClient with this additional development costs.

In summary, BusyClient has to take decision: is it more costly to have Mr. Seller work for a while at AgileCoders or to pay more for SuperTool?

BusyClient can decide using the concept of risk exposure explained previously. The risk exposure of losing business opportunities increases the more Mr. Seller is absent from BusyClient. On the other side, the risk exposure of an expensive development of SuperTool decreases with the amount of time Mr. Seller invests to manage requirements at AgileCoders (see Fig. 6.2). BusyClient should consider the total risk exposure and choose the sweet spot that minimizes it.

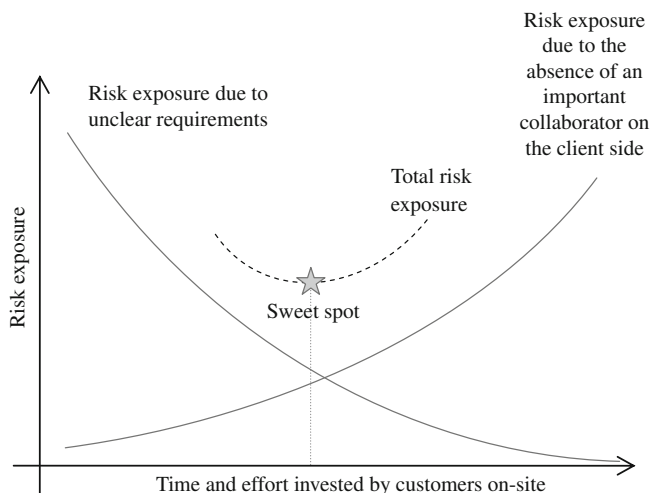


Fig. 6.2 Trade-off between an on-site and off-site customer

The current (2nd) edition of Extreme Programming sees on-site customers (the practice is now called “Real Customer Involvement”) as a corollary practice, i.e., as “difficult or dangerous to implement before completing the preliminary work of the primary practices [4].”

In any case, with or without a customer on-site, AgileCoders and BusyClient have to have a shared view on how the project is carried out. If they do not, BusyClient might expect something different from what AgileCoders is delivering. We observed such a situation with a company that was doing frequent releases. After some time we noticed that the client was quite nervous because he interpreted the

frequent releases as an indicator for low quality: in his eyes, the developers could not get things right and had to fix things continuously.

There are several approaches to communicating a strategy to a client. One possibility is a “mission statement”: it states the vision and describes the chosen means to achieve it.

The mission statement of St. Michael’s Hospital in Ontario states that its vision is “Creating a healthier world, through our culture of caring and discovery” and states the following means to achieve it [29]:

1. providing exemplary physical, emotional, and spiritual care for each of our patients and their families;
2. balancing the continued commitment to the care of the poor and those most in need with the provision of highly specialized services to a broader community;
3. building a work environment where each person is valued and respected and has an opportunity for personal and professional growth;
4. advancing excellence in health services education;
5. fostering a culture of discovery in all of our activities and supporting exemplary health sciences research;
6. strengthening our relationships with universities, colleges, other hospitals, agencies, and our community; and
7. demonstrating social responsibility through the just use of our resources.

The mission statement is easy to understand. Its aim is to be a general guidance for the day-to-day decisions within the organization.

The problem arises if we do not know if the mission is being achieved or not. It is like not knowing where we are on the map, then we do not know where to go to reach our destination.

To understand how good we are in achieving the mission, we need to find ways to measure it. For example, for the point five of the mission statement above, we could look at the “Percent of time dedicated to research.” This measurement would tell us how much time employees are able to dedicate to research.

Only through measurements can we objectively (see box below) assess the current situation and compare our performance with the performance of others or with our performance of the past.

Using the “Percent of time dedicated to research” to measure point 5 of the mission statement defines what we specifically mean by it. It shows what we consider important but also what we do not. For example, by not measuring tangible results like patents or papers, we tell that we do not consider them essential.

This example shows that to find the right set of measurements, we need to have a clear understanding of what is causing success and what is preventing it. If we have a wrong perception of the reality, we will measure the wrong thing.

In the city, to measure how much time it will take us to reach some place, it is fine to use the distance in km. On the mountain this is not enough. We have to consider the height difference too; otherwise our estimation will be very imprecise.

This example shows that it can be necessary to collect a set of measurements to have a precise understanding of the situation. On the other hand, we prefer having few measurements to explain a situation than to have many. This preference (in statistics called “parsimony”) aims to keep the measurement easy to understand (and easy to extend if needed).

What does “objectively” mean?

The term “objectively” is a word used in everyday’s language. Objective is the opposite of subjective. It means that we try to observe some object excluding the influence of us looking at it. This is sometimes difficult or even impossible.

For example, if we take an experienced skier and ask whether some skiing slope is steep, he will probably say: “no.” If we ask a beginner, he might be frightened just to think about it.

The two answers are subjective: they depend on who gave the answer. We cannot compare the answers of many skiers, since they are based on evaluations of the terrain that are influenced by their own experience.

To get an objective answer, we need to find a way to measure the steepness of the slope, independently from who is measuring. We need to (a) define a measure of steepness and (b) define how the measurement is obtained, i.e., define a measurement procedure.

The second aspect—to define the measurement procedure—is crucial: only if the measurement can be performed by anybody obtaining the same result can we then speak about an objective measurement.

We could define the steepness in percent as the relationship between the vertical climb and the horizontal distance. We measure how much height a slope gains in relationship to how much horizontal distance it gains. Figure 6.3 shows a slope where (at the point where we measured) the vertical climb is 0.88 m and the horizontal distance is of 2 m.

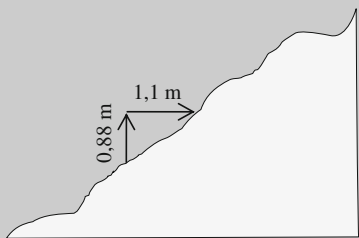


Fig. 6.3 Measuring the steepness of a slope

(continued)

According to our definition, the steepness in percent is calculated as

$$\frac{0.88 \text{ m (vertical gain)}}{1.1 \text{ m (horizontal distance)}} \times 100 = 80 \%$$

As a measurement procedure we choose to use two yardsticks: one is positioned perpendicular and is used to measure the vertical gain, and one leveled yardstick is used to measure the horizontal distance from the end of the first yardstick back to the slope.

Using the steepness in percent and the agreed measurement procedure, we can objectively say how steep a slope is, whether it is 10 % or around 100 % as the couloir of Fig. 6.4.



Fig. 6.4 Joel couloir, Sella group, Dolomites, Italy: Is it steep or not?

An example of a “mission statement” with measurements is the Balanced Scorecard (already mentioned in Chap. 3). The goal of the Balanced Scorecard is to provide a balanced (all aspects of the company should be considered) view of the performance of the company. The Balanced Scorecard itself, in its entirety, can act as a mission statement since it defines what is important (what is measured, what is considered relevant in the organization) and what is not.

The Balanced Scorecard is structured in perspectives, which are the different views of the organization. The initial set of views proposed by the authors are [17]:

1. **Customer perspective:** measures the ability of the company to provide value to the customers. This perspective includes performance, quality, and service measurements.
2. **Internal business perspective:** measures the ability of the company to adapt the internal processes to satisfy customer needs.
3. **Innovation and learning perspective:** the customer and internal business perspective define what the company considers important for competitive success. For example, the ability of the company to innovate, improve, and learn.
4. **Financial perspective:** measures if the company’s strategy, implementation, and execution are contributing to bottom-line improvement.

The Balanced Scorecard helps to get an overall picture of the company.

A problem affects different parts of the company at different times. For example, a customer service that is not able to satisfy customers will be visible looking at the internal business perspective. If customers complain, it will appear in the customer perspective. Finally, if customers switch to the competition, we will see it in the financial perspective.

This means that we can map cause and effect relationships within the Balanced Scorecard [20]. Figure 6.5 shows some of them as arrows between the perspectives.

We now present two ways to communicate a strategy to stakeholders: the mission statement as well as the Balanced Scorecard. These two examples differ in the approach: the Balanced Scorecard is a quantitative approach that collects quantitative evidence to interpret the reality; the mission statement follows a descriptive, qualitative approach (see Chap. 11) to give an overall picture of the elements that characterize the strategy and how they interact.

The way a strategy is described depends also on the type of control we want to exert. In Chap. 4 we discussed behavior controls (e.g., to ensure that employees dedicate a certain amount of time to research) and outcome controls (e.g., to ensure that developers produce a certain amount of code per year). In the same way, a strategy can define the desired behavior and/or the desired results qualitatively or quantitatively.

There is decadelong debate whether the qualitative or quantitative approach is preferable [3]; both have their advantages. Some authors combine qualitative and quantitative approaches, for example, as “exploratory designs” or “explanatory designs” [6]. An exploratory design begins with a primary qualitative phase, then the findings are validated by quantitative results. An explanatory design is characterized by an initial quantitative phase that is followed by a qualitative phase. Usually, the qualitative results serve to explain the quantitative results.

In Lean management we find qualitative and quantitative approaches. The standard worksheet (see Chap. 2) uses a qualitative, descriptive approach. Autonomation uses a quantitative approach to detect a problem; it requires some measurable property to verify its correct value. Because of the importance of autonomation in Lean management, we focus on quantitative ways to define and evaluate the achievement of strategies.

We will use the GQM approach described in the next chapter (similar to the Balanced Scorecard approach, but more general) to quantitatively define the common vision, i.e., what Lean really means for the company. This will alleviate the problem of communicating the goals and methods of Lean to stakeholders and build trust towards those that claim the advantages of Lean.

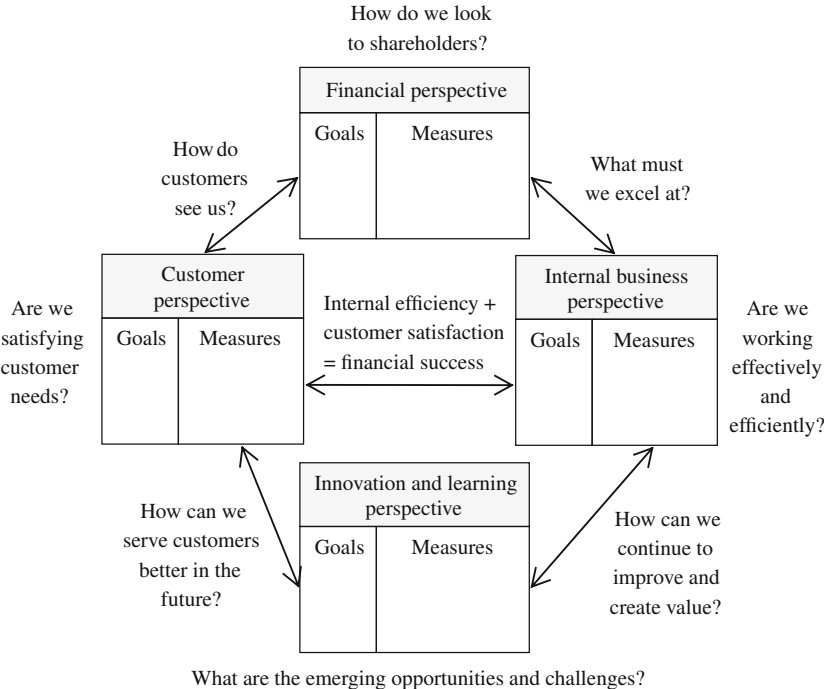


Fig. 6.5 The Balanced Scorecard [17]

6.4 Deprive Gurus of Their Power

We previously stated that Agile methods have been conceived and refined by “gurus.” What we criticize is that gurus tell us the “know-how,” but not the “know-why.” This critique is not completely fair, since gurus not always actually know the “know-why,” i.e., the reason why what they preach is working.

Frequently, the gurus were those people that discovered the new method (e.g., Ken Schwaber, Kent Beck). They made the experience that something works and something does not. It is this experience that they are describing in their books. This does not mean that they were able to develop the wisdom why their method works.

If we need to find out how good a certain technology can work for us and cannot find anyone that can tell us, we have to develop the experience ourselves. We need to use the so-called scientific method¹ to systematically find the knowledge we seek. We need to [3] (see also [32]):

1. formulate a problem in form of hypotheses, i.e., tentative explanations;
2. identify what we want to study;
3. apply research methods to obtain data (e.g., observation, survey, experiment);
4. analyze the data; and
5. use the results to confirm or falsify the hypotheses;

Usually the scientific method begins with idea that pops up or somebody promoting a new technology or method to us. “You have to do testing, then you will have software without defects!” might be a claim. If you are a risk-seeking person, you will immediately introduce testing throughout the company. You risk that the advice is wrong and that the defects increase or that other aspects (such as development speed) suffer. If you are a risk-averse person, you follow the scientific method.

According to the scientific method, we need to formulate the research problem first. An initial formulation could be: “Does testing reduce the number of defects?” We will begin to investigate this question, develop test cases for classes, and document the defects that we find for both types of classes: tested and untested.

We then will formulate the hypothesis: “Testing a method reduces the number of defects in that method.” Counting the defects that we find and classifying them whether they were found in tested or untested classes is the measurement with which we test our hypothesis.

There is a difference between confirming hypotheses and falsifying them. If we confirm a hypothesis, we do not know if there is some situation in the future that falsifies it. If we falsify a hypothesis, we know it is false. If we are not able to falsify it, we can consider the hypothesis provisionally valid.

¹The here described scientific method should not be confused with the scientific method promoted by Frederick Winslow Taylor.

In our case the result of our experiment can have three results:

1. we proof that the hypothesis is wrong: we now know that—in our environment and in our experimental setting—testing does not reduce the number of defects;
2. we cannot proof that the hypothesis is wrong: we now know that—in our environment and in our experimental setting—testing can reduce the number of defects; and
3. we cannot proof that the hypothesis is wrong or right (e.g., if the results are random): we cannot say anything; we have to continue investigating. Using the words of the English writer William Cowper (1731–1800): “Absence of proof is not proof of absence.”

The example shows that a hypothesis that is formulated vaguely is hard to falsify. It is difficult to proof that testing never reduces the number of defects present in a method. Moreover, the usefulness of a hypothesis that could not be falsified is not high: the statement that testing can help is not that useful. An example of a more specific hypothesis is: “Does the % of code that represents testing code correlate with the number of defects?” Failing to falsify this hypothesis would mean that the more we test, the less defects we have. We could then start to look at the optimal amount of testing, and so on.

The experience that we gain from our experiments should be used to improve our work. Only then the time and effort we invested will pay off. We have to document our findings and refine them as we get more knowledge. To be able to apply it and to get the support from others, we need to communicate our findings in way that others understand it.

If we want to share our experience, we have to package it in a reusable form. Reusable means that the know-how and know-why become evident: others can understand how and why it works. Experience reuse wants to make use of previously gained experience in similar problems to help to solve an actual one.

Being able to use a previously packaged experienced has several advantages [5]:

- **Shorter problem-solving time:** the cumbersome task of designing experiments, formulating hypotheses, collecting data, etc. can be avoided.
- **Improved solution quality:** building on previous experience can reduce the probability of wrong decisions.
- **Less skills are required:** the problem solver needs to have less skills if he can rely on previous experience.

The application of the scientific method in software development—the continuous experimenting, evaluating, and adapting—is the way how software companies innovate and gain competitive advantage. This finding convinced scientists and practitioners to develop process models that embed the steps advised by the scientific method.

Lean Thinking also advocates this method to constantly improve.

An example of such a study would be to compare Kanban and Scrum in a specific context and find out their different effects. Such a study was conducted by Sjøberg et al. [28], and they discovered that in their context, after replacing Scrum with their implementation of the Kanban concept, they were able to reduce the number of bugs and improve productivity.

In Chap. 8 we will look at the Experience Factory, which is one way to perform continuous improvement using the scientific method and builds on the Plan-Do-Study-Act method presented previously.

The relationship between Lean and Agile

We can find Agile ideas within Lean Thinking: “Everyone knows that things do not always go according to plan. But there are people in the world who recklessly try to force a schedule even though they know it may be impossible. They will say ‘it is good to follow the schedule’ or ‘it is a shame to change the plan,’ and will do anything to make it work. But as long as we cannot accurately predict the future, our actions should change to suit changing situations [21].”

Here Ono clearly describes Agile practices within the Toyota context. But in the Toyota Production System, Agility is a means to an end, not an end in itself. Also inside Agile Methods there are Lean principles, for example, the tenth principle of the Agile Manifesto: “Simplicity—the art of maximizing the amount of work not done—is essential.”

Altogether, these methods use each other to achieve their respective goals:

- Agile Methods aim to achieve Agility, i.e., the ability to adapt to the needs of the stakeholders.
- Lean production aims to achieve efficiency, i.e., the ability to produce what the stakeholders need with the least amount of resources possible.

Both methods, Lean production and Agile methods, focus on being effective: to maximize the value for their stakeholders. However, they have different perspectives. While Agile Methods focus on software development, Lean production is an approach that aims to optimize the entire organization.

To illustrate the different perspectives of Lean and Agile, we look at the entire “socio-technical system” (see Fig. 6.6).

(continued)

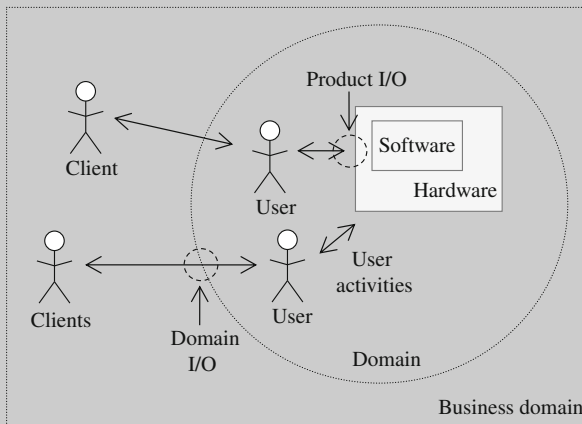


Fig. 6.6 Socio-technical system [18]

This concept looks at IT organizations from three perspectives:

- the product perspective is about the hardware and software of the product;
- the domain perspective is about how users use the product; and
- the business domain perspective is about the business value the users are able to add using the product.

For example, in most restaurants nowadays the waiter uses a device to register the orders. The device is the product. The waiter is the user of the product. The business domain perspective analyzes how well the waiter is able to satisfy the wishes of the guests using the product.

Software is embedded into a socio-technical system, and users interact with hardware and software to solve issues, which help to fulfill their business goals. Therefore, software cannot be seen as a purely technical issue [27].

The difference between Agile and Lean is that they were conceived to work in different perspectives of the socio-technical system.

Agile methods concentrate on the delivery of a product that provides value to the user. The point of view is the one of the developer creating a product for the user. The user knows what is best for him and provides the requirements.

Lean Thinking looks at the entire business domain and seeks the most efficient way to create value for the client of the organization, not the user of the product. This allows to optimize over the entire organization, not only within the activity of software development as Agile methods do.

6.5 Disarm Extremists

In contrast to the gurus of the previous chapter, Agile extremists are the followers of the guru. The extremists we are talking about are risk neutral, optimistic, and idealistic people.

They are willing to accept risk to introduce radical changes. Because of their optimism, risk is not managed, i.e., anticipated, estimated, and minimized using countermeasures, but it is ignored. Problems are addressed as they arise. Their idealism makes them see the whole world as Agile, Lean, etc. Every problem is framed in their “believe system,” in their view of the world.

To rise the awareness that a given technology does not always work, we need objective data; otherwise, we and the extremists discuss based on faith.

Unfortunately, the collection of objective data is expensive. The costs to introduce a measurement program (for the first year) can account for 1–2 % of the total engineering or IT effort [9]. In a study Rico and Pressman made in 2004, the complete cost to use a manual measurement program like the Personal Software Process [14] to help produce 10,000 lines was \$145,600 [24].

To disarm extremists and confront them with hard data, in Chap. 8 we introduce non-invasive measurement. This term—borrowed from medicine—indicates that the measured object is not altered because of the measurement. In the case of measurement, the term indicates that we adopt an approach in which no time has to be spent for the measurement itself, just for the data analysis and interpretation. This kind of measurement is non-invasive because it does not disturb, i.e., distract those involved in the measurement process.

6.6 Summary

This chapter is an anticipation of what will follow in the following chapters: we will introduce the different components we propose to create what we describe in the preface: a practical implementation of Lean software development, gluing together well-proven tools to provide a way to develop Lean. We want to achieve this through the utilization of goal-oriented, automated measurement for the creation of a Lean organization and the facilitation of Lean software development.

The components we foresee are:

- Agile Software Development, described in Chap. 4,
- Non-invasive measurement, described in Chap. 9,
- GQM⁺ Strategies, described in Chap. 7,
- the Experience Factory, described in Chap. 8, and
- Lean Thinking (together with the practices proposed by Taiichi Ono in his book “The Toyota Production System”), described in Chap. 2.

In Chap. 10 we will see how the different components work together.

Problems

6.1. Tag each software development practice of Mary and Tom Poppendieck's proposal of Lean software development as:

- **value:** if its primary goal is to identify what has value and what has not;
- **knowledge:** if its primary goal is to increase the understanding of what happened, what is happening, and what will happen; and
- **improvement:** if its primary goal is to improve the status quo.

6.2. Imagine you have to develop a Balanced Scorecard for a software development team. Which perspectives would you use? Which goals would you use for each perspective?

References

1. Apache Software Foundation: Apache ant (2013). Online: <http://ant.apache.org>. Accessed 4 Dec 2013
2. Apache Software Foundation: Apache subversion (2013). Online: <http://subversion.apache.org>. Accessed 4 Dec 2013
3. Atteslander, P.: Methoden der empirischen Sozialforschung. Studienbuch Series, 10th edn. Walter de Gruyter, Berlin (2003)
4. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley, Reading (2004)
5. Bergmann, R.: Experience Management: Foundations, Development Methodology, and Internet-Based Applications. Lecture Notes in Computer Science. Lecture Notes in Artificial Intelligence, vol. 2432. Springer, Berlin (2002)
6. Borrego, M., Douglas, E.P., Amelink, C.T.: Quantitative, qualitative, and mixed research methods in engineering education. J. Eng. Educ. **98**(1), 53–66 (2009)
7. CppUnit Contributors: Cppunit—c++ port of junit (2013). Online: <http://sourceforge.net/projects/cppunit>. Accessed 4 Dec 2013
8. CruiseControl contributors: Cruisecontrol (2013). Online: <http://cruisecontrol.sourceforge.net>. Accessed 4 Dec 2013
9. Ebert, C., Dumke, R.: Software Measurement: Establish, Extract, Evaluate, Execute. Springer, Berlin (2007)
10. Fowler, M.: Continuous integration (2006). Online: <http://martinfowler.com/articles/continuousIntegration.html>. Accessed 4 Dec 2013
11. Free Software Foundation: Gnu make (2013). Online: <http://www.gnu.org/software/make>. Accessed 4 Dec 2013
12. GIT Contributors: Git (2013). Online: <http://git-scm.com>. Accessed 4 Dec 2013
13. Hibbs, C., Jewett, S.P., Sullivan, M.: The Art of Lean Software Development: A Practical and Incremental Approach. Theory in Practice. O'Reilly Media, Sebastopol (2009)
14. Humphrey, W.S.: Introduction to the Personal Software Process. Addison-Wesley Professional, Reading (1996)
15. Jenkins CI Contributors: Jenkins ci (2013). Online: <http://jenkins-ci.org>. Accessed 4 Dec 2013
16. JUnit Contributors: Junit (2013). Online: <http://sourceforge.net/projects/junit>. Accessed 4 Dec 2013
17. Kaplan, R.S., Norton, D.: The balanced scorecard: measures that drive performance. Harv. Bus. Rev. **70**(1), 71–79 (1992)

18. Lauesen, S.: *Software Requirements: Styles and Techniques*. Addison-Wesley, Harlow (2002)
19. Maglyas, A., Nikula, U., Smolander, K.: Lean solutions to software product management problems. *IEEE Softw.* **29**(5), 40–46 (2012)
20. Martinsons, M., Davison, R., Tse, D.: The balanced scorecard: a foundation for the strategic management of information systems. *Decis. Support Syst.* **25**(1), 71–88 (1999)
21. Ōno, T.: *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, Cambridge (1988)
22. Oracle: Virtualbox (2013). Online: <http://www.virtualbox.org>. Accessed 4 Dec 2013
23. Poppendieck, M., Poppendieck, T.: *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley Professional, Upper Saddle River (2006)
24. Rico, D.F.: *ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers*. J Ross Publishing Series. J. Ross Publishing, Boca Raton (2004)
25. Shalloway, A., Beaver, G., Trott, J.R.: *Lean-Agile Software Development: Achieving Enterprise Agility*. Lean-Agile Series. Addison-Wesley Professional, Upper Saddle River (2009)
26. Sikuli Contributors: Sikuli script (2013). Online: <http://sikuli.org>. Accessed 4 Dec 2013
27. Sitter, L.U.D., Hertog, J.F.D., Dankbaar, B.: From complex organizations with simple jobs to simple organizations with complex jobs. *Hum. Relations* **50**, 497–534 (1997)
28. Sjøberg, D., Johnsen, A., Solberg, J.: Quantifying the effect of using kanban versus scrum: a case study. *IEEE Softw.* **29**(5), 47–53 (2012)
29. St. Michael's Hospital: St. Michael's Hospital, Mission & Values (2013). Online: <http://www.stmichaelshospital.com/about/mission.php>. Accessed 4 Dec 2013
30. van Deursen, A.: Customer involvement in extreme programming: Xp2001 workshop report. *ACM SIGSOFT Softw. Eng. Notes* **26**(6), 70–73 (2001)
31. VMWare: vmware (2013). Online: <http://www.vmware.com>. Accessed 4 Dec 2013
32. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: *Experimentation in Software Engineering*. Computer Science. Springer, Berlin (2012)

<http://www.springer.com/978-3-662-44178-7>

Lean Software Development in Action

Janes, A.; Succi, G.

2014, XV, 393 p. 188 illus., 63 illus. in color., Hardcover

ISBN: 978-3-662-44178-7