Chapter 2
Fundamental Concepts of Knowledge Management

Chapter 1 provided an overview of the basic terms and goals of experience and knowledge management. Tacit, implicit, and explicit knowledge were distinguished. The basic concepts of data, information, and knowledge were introduced and discussed. A layered model for knowledge transfer was built upon those terms. In the end, the benefit of a number of knowledge management interventions in software engineering situations was evaluated. In Chap. 2, we will look a little deeper into the theoretical foundations of knowledge management. This will provide the background for the remaining chapters.

2.1 Objectives of this Chapter

After reading this chapter, you should be able to:

- Explain iterative models of learning.
- Sketch a typical knowledge management life-cycle and point out its challenges.
- Explain the relationships of individual, group, and organizational levels in learning and knowledge sharing.
- Identify software engineering situations where knowledge management can make a contribution and explain how value can be added in those situations.
- Recall the structure and outline of the Software Engineering Body of Knowledge (SWEBOK) [56].

This chapter refers to the overview given in Chap. 1. It details some of the core concepts, such as learning, organizational levels, and adequate application scenarios. Underlying theories of the above-mentioned issues are fundamental to knowledge engineering in general. The SWEBOK catalogue of software engineering knowledge [56] is a foundation for identifying learning topics in software engineering.

Recommended Reading for Chap. 2

- Nonaka, I. and T. Hirotaka, *The Knowledge-Creating Company*. 17 ed. 1995, Oxford: Oxford University Press

K. Schneider, *Experience and Knowledge Management in Software Engineering*,
DOI 10.1007/978-3-540-95880-2_2,

- Argyris, C. and D. Schön, *Organizational Learning: A Theory of Action Perspective*. 1978, Reading, MA: Addison-Wesley
- Argyris, C. and D. Schön, *Organizational Learning II: Theory, Method and Practice*. 1996, Reading, MA: Addison-Wesley
- Schön, D.A., *The Reflective Practitioner: How Professionals Think in Action*. 1983, New York: Basic Books
- Johnson-Laird, P.N., *Mental Models*. 1983, Cambridge: Cambridge University Press
- Fischer, G., *Turning breakdowns into opportunities for creativity.* Knowledge-Based Systems, 1994. 7(4): pp. 221–232.

2.2 Learning Modes and the Knowledge Life-Cycle

Learning and knowledge are two central concepts of knowledge management. There are a number of very well known and influential theories on learning and knowledge. We will first look at Argyris' and Schön's seminal work on loops in learning [7, 8]. It explains the iterative character of learning, which is also inherent in knowledge management. Schön puts an emphasis on reflection. His work on reflection-in-action [98] has a major impact on practical approaches to learning in a working environment. The work by Nonaka and Takeuchi [77] reacts to Argyris and Schön and to many other sources. It is well known for its view on the tacit-explicit dichotomy in knowledge modes. Implications of those concepts lead to a generic knowledge life-cycle. It can serve as a reference model for more software-specific and tailored variants of knowledge management processes.

Some further related work will be put into perspective:

- We will take a glance at the logical patterns of deduction, induction, and abduction that are the theoretical foundation of reasoning with experience and knowledge.
- Mental models are not only a theoretical concept but also a background and driver of some practical techniques described in later chapters.
- Learning is foremost an individual activity. Organizational learning has several aspects that transcend individual learning, but it will only work with employees who are willing and able to contribute to learning at the workplace.
- A look at Popper's famous philosophical work on theories and their merits will close this section. His argumentation is of general value for everyone working with knowledge, theories, and experiences of unknown credibility.

2.2.1 Loops in Learning: Argyris and Schön

There are numerous theories on learning in general. Everyone studying knowledge management or learning issues needs to know the core of Argyris' and Schön's concepts of single-loop and double-loop learning. In general, they see learning as a cyclic process.

Definition 2.1 (Governing variables)
There are governing variables that determine the goals and constraints of acting in a certain situation.

An "**action strategy**" is derived from those governing variables. It leads to intended or real *activity*. When those actions or activities are carried out, certain "consequences" follow. Learning occurs by comparing consequences of actions with the desired consequences of the *action strategy*, which are dictated by the governing variables. If there is a deviation between planned and perceived consequences, the action strategy will be modified. The intention is to better meet goals and governing variables and, thus, to reduce the deviation of actual from desired consequences.

It is important to note that this loop does not necessarily need to be carried out in reality. Both actions and consequences may also occur in the minds of knowledge workers, or in a simulation model, to name just two alternative options. Therefore, a learner does not need to make all kinds of mistakes: It is sufficient to imagine making them. Real or imagined feedback on the outcome will lead to an improved action strategy. Governing variables stay unchanged and provide the criteria by which "improvement" is being evaluated. Learning is considered a cyclic process of reducing deviations in consequences. Argyris and Schön call this process "single-loop learning," as there is a single feedback loop of (real, simulated, or imagined) consequences back to the actions that caused them (Fig. 2.1).

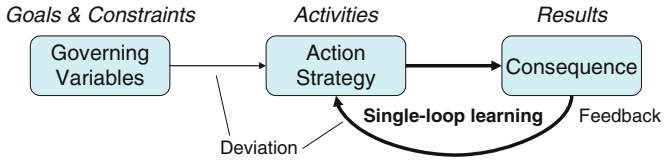

Fig. 2.1 Single-loop learning

Example 2.1 (Single-loop learning in software engineering)

In software engineering, single-loop learning can occur, for example, when a developer writes a program in a language she is not very familiar with. Whenever there is a compiler error, or when tests uncover unexpected behavior, the developer will modify her programming actions to reach her goal of producing a program for a given set of requirements. Constraints include the syntax and semantics of the language, and the requirements for the program constitute the goals.

Feedback is an essential ingredient of learning according to that theory. There is no improved behavior without feedback on (potential or real) consequences of possible actions.

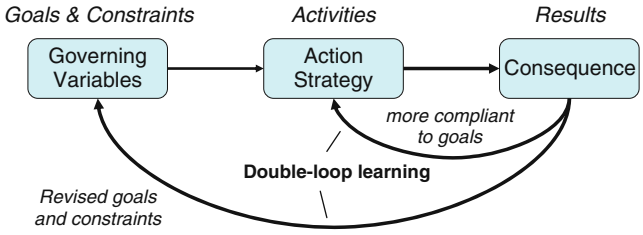

Fig. 2.2 Double-loop learning

Argyris and Schön argue that there is a second mode of learning that transcends single-loop learning. As Fig. 2.2 illustrates, the above loop is maintained but enclosed in a second, outer loop that affects the governing variables. A deviation is again identified by using the governing variables to compare consequences. However, unlike single-loop learning, the outer loop may modify the governing variables: Reflecting on the background and reasons of a deviation may lead to *changing constraints and goals* as well. In single-loop learning, a specific problem is solved by adapting the action strategy. In double-loop learning, a set of governing variables (goals and constraints) is questioned, which may impact many future problems. Knowledge is acquired on that higher level. Both action strategy and governing variables may be adjusted in order to satisfy (adjusted) goals and constraints better.

Example 2.2 (Programmer)

The programmer in the above example conceives a module. If its foreseeable behavior differs from the specified (required) behavior, the programmer either can change the module (single-loop learning) or may challenge the requirements and try to extend the constraints. Maybe there was a misunderstanding? If customers agree to change the requirements, the new requirements may be fulfilled without changing the module. This is an example of double-loop learning. Corrected requirements or improved requirements engineering procedures will help to avoid similar problems in the future.

How can people carry out action strategies without acting? The concept of **mental models** helps to better understand what this means. Johnson-Laird [60] defines mental models: "... humans create working models of the world by making and manipulating analogies in their minds." In their recommendations for double-loop learning, Argyris and Schön refer to mental models. Mental models convey the schemata and frameworks for anticipating the outcome of actions. Therefore, Argyris and Schön use *maps* (their word for mental models) to find common ground with decision makers and knowledge workers.

Just for exercise, we can interpret Fig. 1.8 as such a map: It provides a simplified overview of a complex piece of reality, namely knowledge management. By using that map as *governing variable*, one may develop an *action strategy* for setting up

a knowledge management initiative. For example, the map in Fig. 1.8 uses dashed lines to indicate direct communication during experience elicitation. It thus encourages us to consider interviews, meetings, or other forms of direct communication rather than document templates. Single-loop learning within the limits of that mental model might imply asking programmers to call in instead of writing. In double-loop learning, the assumption of the mental model is challenged: Maybe, under certain circumstances, a phone call is less appropriate than a written note. This will change the map and the governing variable it represents, adding a solid line for written experience reports. By taking that possibility seriously, the initiative may *encourage written complaints* as yet another form of urgent, emotionally intense kind of experience. This transcends the initial goal and modifies the mental model we started with. In both cases, the deviation between assumed and real outcome has been reduced. Either the outcome or the assumptions has been adjusted.

Mental models provide guidance for our actions and plans. Maps like Fig. 1.8 can be seeds for mental models. By conveying a lot of information, a single map can guide single-loop learning and become an initial reference for double-loop learning, too.

2.2.2 A Knowledge Management Life-Cycle

We have already encountered a central topic of Nonaka's and Takeuchi's theory of knowledge creation [77] in Chap. 1: the dichotomy of tacit and explicit knowledge. In their theory, they considered the work by Argyris and Schön but extended it to a three-dimensional life cycle of knowledge management.

The first dimension of knowledge creation is the tacit–explicit dimension. In Fig. 2.3, those two modes are shown twice: The matrix describes the four possible **conversions** of the two modes of tacit and explicit knowledge. When tacit knowledge is converted into explicit knowledge, this is called *externalization*, as we saw in

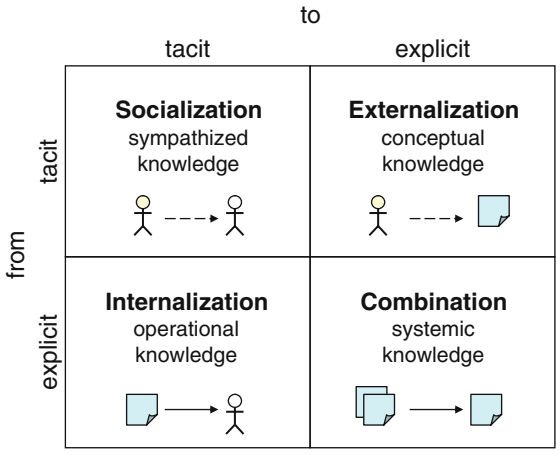

Fig. 2.3 Conversions between tacit and explicit knowledge

Chap. 1. Vice versa, *internalization* stands for converting explicit into implicit and finally maybe tacit knowledge. Simple examples are reading (internalization) and writing (externalization). Usually, internalizing a skill will take more time and more effort than will internalizing a simple procedure. Operational knowledge is typically being internalized: A person reads a recipe and tries to gain cooking knowledge. Often, conceptual knowledge is externalized: From all the tacit knowledge in someone's head, only the concepts (abstractions and simplifications) get documented.

There are two more conversions: We have briefly touched on *socialization* in Chap. 1, indicating the transfer of knowledge from one person directly to another. In socialization, there is no explicit intermediate externalization and internalization as we defined it. However, the consequences of one person acting or speaking (externalization) are observed or heard by the learner (internalization). This is often a very intense but not highly efficient mode of transfer. Making knowledge explicit helps making the transfer of knowledge faster, more reliable, and accessible to a larger group of learners.

Example 2.3 (Combination)

When explicit knowledge is converted into other explicit knowledge, this is called *combination*. For example, a middle manager may combine a company policy with a concrete project budget to instantiate a concrete quality assurance plan for that project. If the contribution of that manager is small compared with the knowledge within those documents, this move is called combination. (Otherwise, we would consider it an invention of "new" knowledge.)

Nonaka and Takeuchi imply an iterative learning process by drawing a spiral in Fig. 2.4. Time progresses along the spiral, and the amount of knowledge learned grows with the diameter of the spiral. It starts with socialization: Someone acquires knowledge by directly observing or talking to someone else. This constitutes a dialogue. After some time, there is a need to write down some of the learned knowledge. By externalizing the key concepts, the documented knowledge will

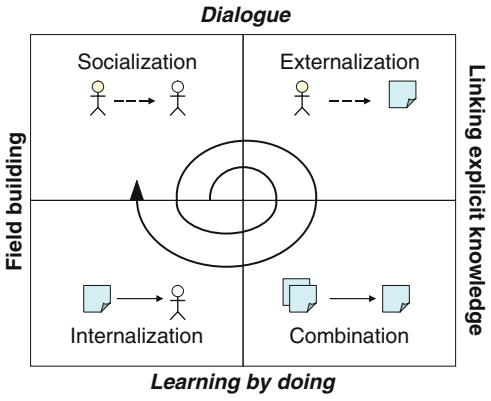

Fig. 2.4 Learning process as a spiral over knowledge conversions

be linked to preexisting explicit knowledge. This is when externalization shifts to combination.

The situation changes again when the knowledge is applied. It turns into operational knowledge, which can be used in actual work. This conversion starts out as learning by applying knowledge in a practical situation (learning by doing). After some experience and exercise, the operational knowledge is internalized better. Gradually, a field of knowledge is built up. The iteration is about to start again. Because of the extended knowledge, the diameter of the spiral grows.

Nonaka and Takeuchi point out several subtle properties of their model:

- They see knowledge as "action, belief, and commitment to an end" [77]. They emphasize the personal relationship over a perspective of knowledge as passive "material." For that reason, knowledge management is not just a matter of logistics but a challenge for learning and creating new knowledge. The opening spiral alludes to creating new knowledge, too.
- Along the same lines, knowledge is created inside an organization and delivered to the outside – not just absorbed and digested outside-in.
- Double-loop learning requires "questioning and rebuilding existing perspectives, interpretation frameworks, or decision premises" [8]. Nonaka and Takeuchi are concerned with managing such a learning process. Who could trigger it, and who could direct it in the sense of managing it? They suggest there needs to be a continuous, ongoing process of learning that includes the rearranging of mental models. Therefore, there is only one spiral that refers to both kinds of learning loops.

The spiral in Fig. 2.4 iterates over the conversion modes. It grows into a knowledge management life-cycle when additional dimensions are added. In Fig. 2.5, not only the tacit/explicit and conversion dimensions are used; this model also refers to the sources and sinks of knowledge. The spiral in the middle of Fig. 2.5 corresponds

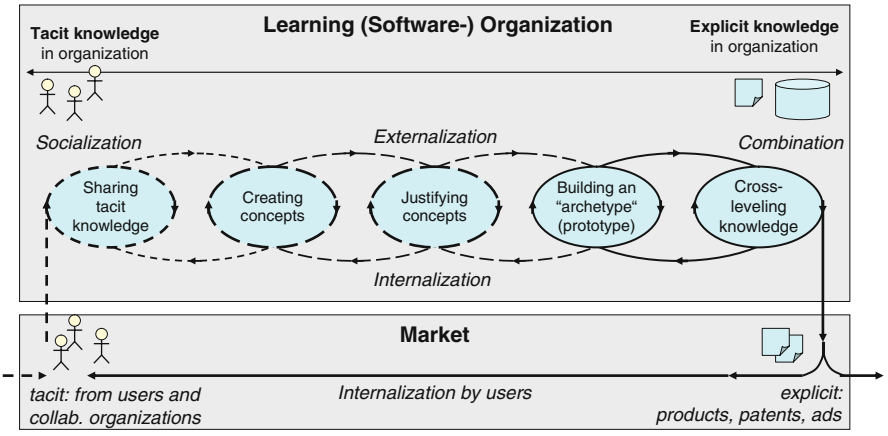

Fig. 2.5 Life cycle of knowledge management according to Nonaka and Takeuchi [77]

with the spiral in Fig. 2.4 when it is stretched along the five stages of maturing knowledge. It is embedded within a larger learning spiral that includes the market. Maturing knowledge is externalized and used in the market as explicit knowledge (patents, products, or services). It is internalized by users. They feed it back as suggestions, complaints, or requirements. Together with supporting knowledge from collaborating organizations, this feedback drives the inner knowledge spiral. Do not confuse the outer spiral with the outer loop of double-loop learning: There are two different user groups intertwined in feedback loops of different speed and granularity. All of them constantly improve both *solutions and mental models*. Dashed lines indicate the flow of tacit and implicit knowledge, and solid lines indicate more explicit knowledge. The small symbols highlight the main media used at both ends of the spectrum. They are people on the tacit end and documents or data stores on the explicit end.

A learning software organization must organize externalization and internalization of knowledge; it also needs to provide opportunities for direct learning via socialization. Life-cycle models like Fig. 2.5 show the conceptual or cognitive phases for sharing knowledge in a company. We are obviously living in a "knowledge society" that depends on learning on all levels. However, Eraut and Hirsh claim:

> Although organisational learning sounds like something the organisation controls, it has become increasingly clear that organisations only truly learn when they give much of that power back to individuals and self-selected groups. [37]

While it is important to keep this warning in mind, we may still look at knowledge logistics and cognitive mechanisms for creating and distributing knowledge. Promoting individuals to responsible drivers of their own learning processes is definitely important for successful learning – and also for organizational learning. In the end, both individual learning commitment and knowledge logistics need to come together.

2.2.3 Kolb's Learning Cycle

A number of core questions can be asked with Fig. 1.8 in mind:

- How can one learn a general lesson from making concrete observations?
- How can conceptual knowledge be applied to a situation at hand? How do we whether it is applicable or not?
- How can widely applicable knowledge be created from a small number of observations someone has made?
- What conclusions can we draw from observing yet another concrete situation?

In his seminal work on learning cycle and learning styles, Kolb [64] claims a "concrete observation" must be the basis of "reflective observation." Reflection is a prerequisite for "abstract conceptualization," as Kolb puts it: Abstract conceptualization is the process of drawing a conclusion from a single or a few observations.

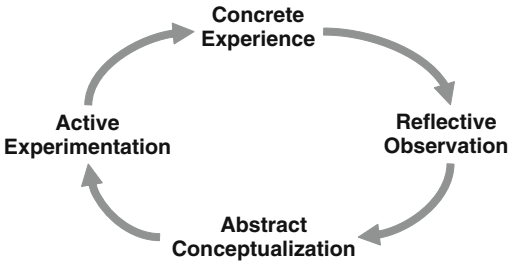

Fig. 2.6 Kolb's learning cycle [64], simplified by Davies [27]

That conclusion will be more general (hence, more "abstract") than the initial observation(s). More abstract and more general conclusions apply to more situations. They can be reused better (Fig. 2.6).

In our terminology, an *experience* includes Kolb's

- "Concrete experience," which corresponds with "observation" in our terms.
- *Results* of "reflective observation" and "abstract conceptualization." The latter results are called *hypothesis* or *conclusion* in our terminology.

Kolb points out that derived conclusions must be used and challenged actively. This will help to validate them – and stimulate making new experiences. Our definition of *experience* includes an emotional aspect. It is missing in Kolb's model. Our definition stresses the difference between an "emotionally neutral" piece of knowledge and an "emotionally loaded" experience. In compliance with Kolb's learning cycle [64], learning from concrete observations and experiences can abstract to conclusions. When such a conclusion is validated and deprived of íts emotional aspect, it may gradually turn into knowledge.

2.2.4 Classical Modes of Reasoning

All theories presented above see knowledge from an action-oriented point of view. Creating knowledge and improving knowledge by iterative learning are closer to psychology than to formal logic. A few short remarks should be sufficient to cover the *logical aspects* of learning and reasoning. They are rooted in **epistemology**, the science of learning and understanding.

Definition 2.2 (Reasoning patterns)
From a logical perspective, reasoning in knowledge engineering follows certain patterns: induction, deduction, and abduction. These concepts provide a guiding structure for all abstractions or applications of knowledge.

Induction: Concluding from one or more specific cases to a general principle. Example: An apple falls to the ground. So does a pear. Induction: All fruits fall to the ground. Further induction: All dead material falls to the ground. Yet another induction: Everything falls to the ground.

Deduction: Concluding from a general principle to a specific case. Example: If all fruit falls to the ground, those cherries will also fall. I cannot know about this pencil (because it is no fruit), but the orange will fall.

Abduction: Inventing a new general principle by deriving a hypothesis from a special case. There are three apples on the ground. I hypothesize that they fell from the tree. I come up with a general rule to explain the one case I have seen (Fig. 2.7).

The patterns deserve some discussion. Obviously, induction is very powerful – but may lead to false theories or conclusions. In the above-mentioned examples, the last induction is not true: Our sun does not fall to the ground of the earth. As we will see, however, practical knowledge management cannot succeed without induction: Inducing new principles is a characteristic of double-loop learning.

The difficulties in the examples above were not due to subjective or psychological aspects. They are purely logical. Of course, induction is applicable to physical and social phenomena alike – but it assumes a *logical* perspective on both. Statements about personal taste, such as "I like chocolate, so does my friend," can be combined by induction to derive "everyone likes chocolate." This induction obviously went too far: There are many people who do not like chocolate. Induction is a logical operator, not a consensus-building activity.

Mnemonic 2.1 (Deduction)
Deduction, if applied according to the rules, never yields wrong results. It is rather too cautious.

In our example, pencils are not fruit and are, therefore, not covered by our knowledge of falling fruit. We may not deduce a pencil will fall. The rules only allow deducing cases that are covered by the general principle. Here lies a challenge for deduction in practice: What exactly is covered by the general principle? Do only apples fall – or also pears, and maybe pencils? This question is easy to answer in formal environments but sometimes very difficult in practice.

Whereas induction and deduction go back to Greek philosophers, abduction was mainly promoted in the 19th century by Charles Sanders Peirce, a proponent of semiotics [52]. Semiotics is the discipline of signs, symbols, and their meaning. Pierce argued for a process of understanding that started with abduction (*I have an idea and generalize it to a principle*), deduction (*if the general principle is true, it*

	Induction	**Deduction**	**Abduction**
Given		$\forall f \in Fruit \Rightarrow f$ falls	
Observed	Apple falls, Pear falls	There is an Orange ($\in$ Fruit)	Apples on the ground
Derived	$\forall f \in Fruit \Rightarrow f$ falls. *may be more general:* $\forall f \in Object \Rightarrow f$ falls	Orange will fall. *I know nothing about pencils.*	Apples fell from the tree and in general: $\forall f \in Fruit \Rightarrow f$ falls

Fig. 2.7 Comparison of induction, deduction, and abduction

applies to these specific cases), and induction (*because the principle was valid with those examples, I assume my principle is correct*).

2.2.5 Reflective Practitioners and Breakdowns: Donald Schön

Donald Schön studied the conditions under which practitioners could create knowledge. In particular, his work on "reflection-in-action" is influential for making knowledge engineering work in practice.

In his book *The Reflective Practitioner* [98], Donald Schön investigated how working and creating knowledge interact. He observed working practitioners and noticed that they hardly reflected on what they were doing *while* they were doing it. While they carried out complex and difficult tasks, they were operating "in tacit mode." Being absorbed by their demanding tasks, they invested all their attention into solving their problem. Afterward, they could not explain what they had done either, because they did not remember in sufficient detail.

Schön found:

Mnemonic 2.2 (Reflection in action)
Interrupting knowledge workers in their task helps them to reflect.

Donald Schön called this a "breakdown" that can lead to reflection and – finally – better understanding. Gerhard Fischer [40, 41] has integrated the concept of breakdowns into the construction of critiquing systems: When a practitioner uses a computer system for a design task, certain situations trigger a critiquing message warning the practitioner. For example, a software design may contain too-deep inheritance of Java classes. A design tool may notice that and remind the architect of the respective design recommendation. This constitutes a breakdown, allowing the practitioner to "wake up" and reflect on the tacit knowledge he has just applied or failed to apply. When you need to externalize tacit knowledge, planning for appropriate breakdowns can help.

2.2.6 Popper: When Should We Trust a Theory?

We have seen the theory of single- and double-loop learning by Argyris and Schön. Schön has suggested eliciting tacit knowledge by generating breakdowns. His theory assumes people will be able to reflect better when they are interrupted. Nonaka and Takeuchi have proposed a sophisticated theory of knowledge creation. In the above section on formal reasoning patterns, we saw a sequence of abduction–induction–deduction, which can produce a theory and test it. Theories abound, plus hypotheses from all experiences! It is obviously important to find out when to trust a theory and when not.

Philosopher Sir Karl Popper is famous for his work on theories [85]. He suggests a good theory must be *falsifiable*: It should be easy to demonstrate that it is false – *if* it is false.

Definition 2.3 (Falsifiable theory)
A theory is falsifiable if it allows making predictions. In addition, it must be easy to recognize when a prediction is violated.

Example 2.4 (Falsification)

For example, a tourist may come up with the theory "it never rains in Southern California." One single day of rain will falsify that theory, which makes it a good theory. As long as it never rains, the theory gains in credibility – but it can never be proved. Even after 100 sunny years, there might be rain eventually. According to Popper [85], this is true for all theories: As long as they are not falsified (despite their falsifiability), theories gain in credibility. As soon as there is one counterexample, the theory is obviously false. This reasoning is very close to the abduction–induction–deduction pattern.

Practical knowledge management relies on experiences and induction for improvement. How can we know that resulting "best practices" are actually better than the original ones? Following Popper, hypotheses gained from experiences (or from other sources) should imply predictions. As long as they hold, this is strong support for the hypotheses. If, however, only a single prediction fails, the hypotheses cannot be true in general. It needs to be either refined or refuted.

Mnemonic 2.3 (Pragmatic use of theories)
In practice, however, one will even stick to theories and hypotheses that have been falsified – as long as no better alternative is available.

2.3 Knowledge in People, Teams, and Organizations

Knowledge management exceeds individual learning. Organizational learning was briefly introduced as a term, and the concept was put into context in Chap. 1. In this section, we will see how the theories by Argyris and Schön and by Nonaka and Takeuchi explain organizational learning *beyond individuals*.

Senge [99] adds a perspective rooted in system theory. The organization is seen as a complex system with many interactions and interdependencies. Other researchers have emphasized the importance of learning for competent behavior. Wenger describes "communities of practice, " and Simon [102] focuses on decision-making by managers. Those well-known approaches were selected to represent the foundations of using and managing knowledge in an organization.

2.3.1 The Scope Dimension in Knowledge Creation

Nonaka and Takeuchi extend their above-mentioned theory on learning into an organizational setting. The dichotomy of tacit and explicit knowledge is the starting point. It represents the dimension of epistemology in their model (Fig. 2.8). Epistemology is the science of knowledge and belief. Tacit and explicit are two different epistemological modes of knowledge. As we have seen above, learning occurs as an

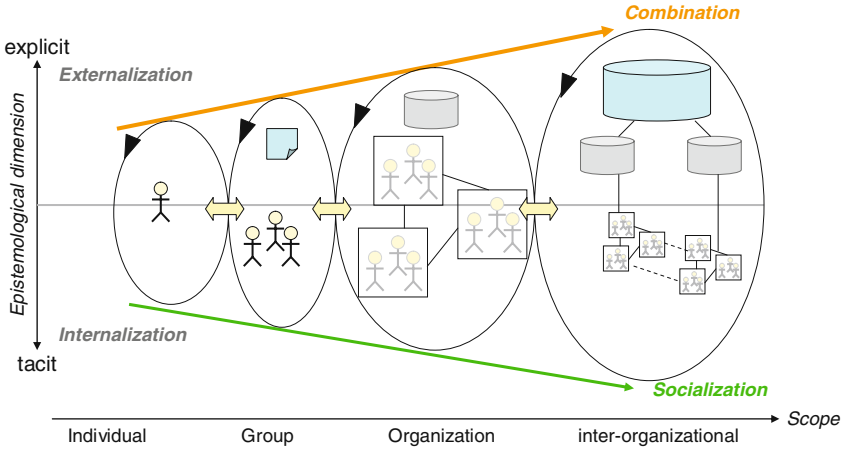

Fig. 2.8 Additional scope dimension of the spiral learning curve with conversions, adapted from Nonaka [77]

iterative process over those modes. The spiral in Fig. 2.4 showed a cycle of conversions between those modes: There is a continuous back and forth between tacit and explicit knowledge.

We will now look at the scope that can be reached through iterative conversions. In a way, the spiral will be stretched over yet another dimension, namely the spectrum of individual versus organizational scope. An individual goes through conversions of tacit and explicit knowledge. By being a member of a team or work group, individual learning feeds into group learning. It cycles through tacit and explicit phases again. A similar iteration occurs on the organizational and even interorganizational levels. There are many intertwined iterations over the epistemological dimension. They can be read from both sides of the scope: Individuals share their knowledge and experience through socialization and combination. At the same time, the cycle of larger units keeps the iterations of smaller units turning.

When looking at Fig. 2.8 from the right-hand side, this scope dimension is invisible. From that perspective, we would only see an oscillation between tacit and explicit knowledge. Fig. 2.5 detailed activities and conversions during this oscillation, thus providing a second dimension. The scope dimension in Fig. 2.8 introduces a third dimension: in Ref. 77 it is called "ontological dimension," but in the context of our current topic, *scope* is a better term.

There are some core messages conveyed by Fig. 2.8:

- The iteration links individuals, groups, and organizations. For example, a sequence of externalizing and then internalizing knowledge is a way to transfer previously tacit knowledge from one person to another, or even to an entire group. Conversions drive the spiral and at the same time allow others to participate and benefit.

- Epistemological modes and conversions occur in organizations and in individuals. Internalization in an organization, for example, may indicate that the organization reacts according to knowledge that is deeply rooted (internalized and tacit) in its individuals, repositories, and infrastructure.
- Knowledge transfer along the scope dimension is not a one-way traffic. Fig. 2.8 shows several individuals and one instance of an organization. The spiral mediates between all of them, and there is knowledge transfer in both directions: Individuals need to receive knowledge to make the organization smarter.

There is a highly complex relationship between individual, group, and organizational knowledge. From a practical point of view, it is most important to remember (1) that it is an iterative process and (2) that it transfers knowledge not just in one direction. Knowledge cycles from tacit to explicit – and back again. And it cycles from the individual to the organization – and back to individuals.

2.3.2 Group Interactions and Shared Maps

Argyris and Schön discuss organizational aspects of learning with respect to their notions of single-loop and double-loop learning modes. For the purpose of knowledge management in software engineering, we will focus on the interrelations between the two learning modes:

As Smith [103] points out, Argyris and Schön consider many organizational learning activities as "Model I" or single-loop learning. However, pure single-loop learning improves behavior only as long as goals and constraints are correct and adjusted to organizational needs. Leaving them unadjusted for a longer time may lead to less advantageous results: Behavior keeps being "optimized" with respect to an outdated measure, which diminishes success.

The inherent **warning** in this argument is as follows: If we support and enforce single-loop learning abilities too much and make it too efficient, its positive outcome will degrade and even turn against the organization. Our goals should not remain static but follow external pressures and demands. Organizations tend to regulate learning (e.g., in knowledge management initiatives) and promote single-loop learning (Fig. 2.9).

Overly efficient single-loop learning may be counterproductive, as the third sketch illustrates: It still hits the old target; but in the meantime, the target has shifted. Double-loop learning helps to adjust goals and targets. It combines elements to better hit a target – and others that help adjusting goals and targets.

Example 2.5 (Adapting or adjusting)

If a software company considers top quality its highest priority, steering projects closer to that goal will be an improvement – for a while. It can be achieved through single-loop learning. But the company needs to notice and to adjust when customers request fast and agile projects more and more often. Sticking to the traditional

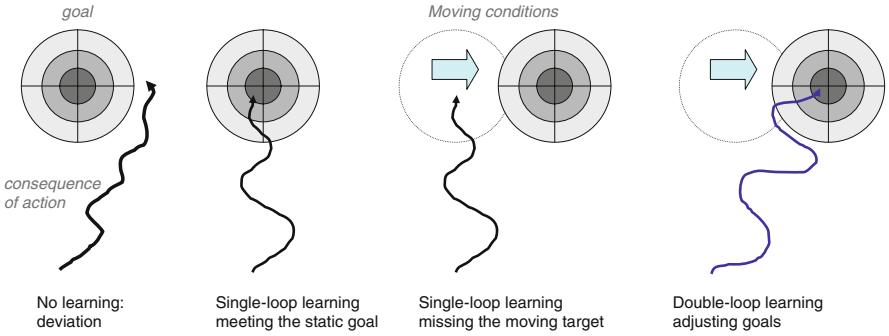

Fig. 2.9 Impact of single- and double-loop learning, with static and moving goals

top-quality goal will prevent people and their projects from learning how to develop in a more agile way.

Mnemonic 2.4 (Adjusting goals)
Outdated goals and "governing variables" prevent learning from adjusting. Shifting goals are more likely to be missed when goals of learning are not adjusted.

Individual software engineers may change their goals and constraints more readily when they are not guided by a company policy ("governing variables"). There is an obvious need for a compromise between guidance toward efficient single-loop learning and opportunity for higher-order double-loop learning.

Productive and nondegrading organizational learning, according to Argyris and Schön, requires double-loop-learning ("Model II"). This setting is characterized by a number of factors – and those need to be promoted by knowledge management, too:

- *People construct maps together*. A map is a mental model that facilitates shared understanding. A map is a schema or model on the "governing variable" level. By revising maps, goals and procedural rules are adapted. When a problem is encountered, there needs to be not only the efficient single-loop option of fixing it but also the double-loop option of rethinking and reframing. Fig. 1.8 can be considered a map in that sense.
- Shared maps constitute shared understanding. Although shared maps will be constantly revised and updated, the process and interaction of constructing a common view is at the core of organizational learning on a double-loop level.
- Not only must a learning organization permit shared construction of maps; there also must be active interventions to encourage group interactions and updating of knowledge repositories. Smith [103] writes:

  For organizational learning to occur, "learning agents," discoveries, inventions, and evaluations must be embedded in organizational memory. [7]

Although Argyris and Schön consider double-loop learning a necessity, they concede it is difficult to reach on an organizational level. Eraut and Hirsh [37] claim

that a different kind of expert is needed on the higher levels where problems are increasingly vague and ill-defined, and an expert's experience mainly helps her to *assess* situations better. Such an expert may not even have superior reasoning skills or problem-*solving* capacities.

Different levels of expertise and experience require different learning environments, incentives, and different techniques. Dreyfus and Dreyfus [33] (cited after [37]) identify five levels of learning that can be related to degrees of (double-loop) goal reflection:

1. Rigid adherence to taught rules or plans (as in single-loop learning).
2. Situational perception still limited, but improved awareness for situation.
3. Standardized and routine procedures at reduced cognitive load.
4. Setting of priorities: Perceives deviations from the normal pattern.
5. Intuitive grasp of situations based on deep tacit understanding.

It is an achievement to learn and follow rules, as on level 1. Single-loop learning is an essential ingredient of a learning organization. It leads to more efficient work at the middle levels. Routine procedures and processes have driven software process improvement over more than a decade with maturity models the like Capability Maturity Model (CMM) [83], its Integrated new version CMMI [29], or the SPICE standard with European roots (ISO 15 504). However, at some point and for some tasks, software engineers need to transcend efficient adherence to given plans. Recognizing patterns is more flexible and calls for more experience. As experience grows, it will become more and more tacit. Gifted software engineers can reach a level of understanding they are not able to explain. It is based on numerous observations and cases and patterns they have seen in their career.

A learning organization does not have to focus on the highest levels of learning only. There are numerous software engineering tasks that call for defined procedures that are communicated and taught well within a business unit. Supporting this level is an honorable and demanding endeavor. Of course, knowledge management should encourage software engineers to assess situations more effectively and, thus, choose the appropriate processes more deliberately.

On the highest level of intuitive and tacit understanding, a knowledge management initiative can still provide information access and infrastructure. At the same time, highly experienced employees will often be knowledge providers rather than knowledge receivers. They turn into the knowledge and experience bottleneck in an organization. Knowledge management can offer them a welcome multiplication and dissemination mechanism to spread what they know to more junior colleagues – at a lower personal effort.

2.3.3 Other Related Theories and Approaches

Nonaka and Takeuchi emphasize different aspects than do Argyris and Schön, but the core of their theories are compatible, at least at the level we need to see in this book. However, there are a few other well-known approaches that influence practical experience and knowledge management in software engineering. They provide

additional pieces for the puzzle that underlies knowledge management initiatives and learning from experience. For that reason, some additional ideas are presented below. We will meet them again in later chapters.

Senge [99] is one of the pioneers of system thinking in organizational learning. He was a student of Chris Argyris, and his bestselling book, *The Fifth Discipline*, shows some common ground with his former teacher. For example, mental models are considered one of the five disciplines Senge describes. The five disciplines are considered prerequisites for organizational learning:

1. *Personal mastery*: An organization learns through its members, and those "parts" of the organization need to develop knowledge, skills, and mastery.
2. *Mental models*: The way we see the world and how it works. Like the above-mentioned maps, mental models are deeply held beliefs, often tacit and sometimes shared.
3. *Shared vision*: An organization can act smarter than each of its parts if it is guided by a common goal.
4. *Team learning*: An organization needs to streamline and bundle the activities and knowledge of its parts.
5. *System thinking*: An organization is a complex and interrelated system of parts and dependencies. More important than optimizing a part is improving the structure of the system.

Senge's main contribution to knowledge management is his strong emphasis on **system thinking**, the "fifth discipline" in the above list. This distinguishes his approach from all others discussed in this section. All authors (including Senge) stress the importance of individual learning for organizational learning. Senge, however, emphasizes the structures of the system, whereas all others underline the social and process-related aspects of organizational learning.

Argyris and Schön talk about loops in learning. Their above-mentioned theory on building organizational maps in double-loop learning is obviously dominated by interaction. Nonaka and Takeuchi depict the process as a (spiral) line. By spiraling through epistemological (tacit vs. explicit) modes and different holders of the knowledge, they describe continuous exchange and interaction. In addition, they highlight knowledge *creation,* which occurs during that process. Nonaka and Takeuchi criticize Senge for focusing too much on individual learning. They advocate creation and development of knowledge as another important source of organizational learning. Our Definition 1.1 (organizational learning) includes collective repositories and a infrastructure as well. The infrastructure includes tools, collaboration opportunities, and processes that guide systematic work in the workplace.

Wenger [119] studied communities of practice as a key part of a learning organization. **Communities of practice (CoPs)** are groups of knowledge workers who share experiences and knowledge in a common field of practice. A CoP is usually a self-organizing group of people that cuts across organizational structures. Unlike a team or organizational unit, members of a CoP do not need to work on the same project or even in the same team. They are volunteer members of the CoP, motivated by their own perceived benefit. Mechanisms of sharing experiences and knowledge within a CoP follow the above-mentioned theories. By not being hierarchically

organized, and by reaching into different parts of an organization, a CoP can facilitate organization-wide learning and spreading of knowledge. Some companies have explicitly founded and encouraged communities of practice as a major component of their knowledge management initiatives, such as Siemens [26]. In many other companies, aspects of cross-cutting volunteer groups are used in different variants. Communities of practice emphasize interactions of individuals.

Dodgson summarizes the structural *and* behavioral aspects of learning organizations [32]: "Learning organizations purposefully construct structures and strategies as to enhance and maximize organizational learning." Obviously, both aspects are needed.

2.4 Software Engineering Knowledge

The foundations of knowledge engineering discussed so far are not specific to software engineering. Because this book is about experience and knowledge management in software engineering, this specific knowledge area is sketched below. This helps students to understand the bigger picture of the examples and discussions throughout the book. Practitioners are reminded of some knowledge-related aspects they will probably know from their own experience. In the remainder of this book, the discussion builds on this selection of software engineering knowledge aspects.

2.4.1 Software Engineering from a Knowledge Perspective

We have seen theories about learning and how knowledge transfer works according to selected famous theories. But what kind of knowledge is worth being managed in software engineering? This depends on what we mean by "software engineering." This term is widely used to refer to any activities related to building software. This intuitive characterization is sufficient for many practical purposes. It can be helpful to define the term more precisely to understand better where knowledge is needed within software engineering.

2.4.1.1 Defining Software Engineering

In 1968, the term *software engineering* was coined. It expressed the desire to turn software development into an engineering discipline, just like electrical or mechanical engineering. Engineers obviously follow a systematic and disciplined process to achieve their results. In computer science, many developers saw themselves rather as artists than as engineers. They emphasized the creative aspect of building software, whereas the engineering perspective emphasized predictability of duration and cost, reuse, and quality-oriented behavior. Software engineering encompasses both aspects. There are many activities that require creativity, knowledge, and experience. At the same time, many critical tasks can be carried out with a systematic and

disciplined approach, thus harvesting the benefits of engineering wherever possible.

According to IEEE Standard 610.12 – 1990, software engineering and software are well-defined terms:

Definition 2.4 (Software Engineering; SE)

1. *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.*
2. *The study of approaches as in (1).*

This definition includes operation and maintenance along with development of software. The scientific approach to study and improve those aspects is also included under the term "software engineering." Many people associate software with program code. However, it does not cover the entire meaning of the term, according to IEEE Standard 610.12 – 1990:

Definition 2.5 (Software; SW)
Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

Documentation, data, and procedures required to install, configure, and operate a computer program are part of software, too. From a knowledge perspective, a substantial amount of technical and application knowledge is represented in manuals, technical documentation, and configuration parameters. All of those are part of software, and software engineering is concerned with acquiring that knowledge and guiding it into program and associated material. Because software engineering targets a systematic and disciplined approach for development, the required knowledge must not be taken as a given – in many projects, most of the application domain knowledge must be acquired or built during the project. Practitioners know that disciplines such as management, psychology, and all the application domains of the software products affect the development of software. There is much more knowledge to handle in software engineering than one may think at first glance.

It is advantageous to share a common understanding of fundamental knowledge challenges relevant to "typical software engineering tasks." We cannot provide a complete or very detailed enumeration of those tasks and challenges here. Instead, pointing to some examples of knowledge-intensive tasks is supposed to create a common basis for both experienced practitioners and students of software engineering.

Tasks will be briefly introduced. This introduction stresses the mission and purpose of tasks, and it highlights their relationships with each other. In particular, related knowledge is emphasized. This short overview provides motivation for considering this task as a rewarding application area of knowledge management. In the next section, those tasks are summarized within the Software Engineering Body of Knowledge (SWEBOK). SWEBOK presents the essence of knowledge-related software engineering tasks. It can serve as a reference model. All examples in this

book are largely self-explanatory. However, most of them refer to situations covered in the following summary. They highlight selected aspects of the bigger picture of software engineering.

2.4.1.2 Core Activities: Requirements, Design, and Software Construction

Software engineering is mostly associated with programming. Writing code in a programming language is a core task. In addition, manuals and technical documentation must be written. According to Definition 2.5, "associated data and documentation" are part of software. Knowing syntax and semantics of a programming language is only a small fraction of the knowledge required in software construction. The ability to transform a given algorithm to a computer program is a skill often acquired during computer science education. An industrial environment requires programmers to adhere to standards and guidelines. Standards come from external sources and regulate the use of techniques that have proven useful. Guidelines and conventions, such as the Sun code conventions (http://java.sun.com/docs/codeconv/), are internal regulations. Coding conventions tell programmers how to comment and format code, how to name variables and identifiers, and how to structure programs in packages, classes, and methods. Developers need to know what relevant standards, guidelines, and conventions are and what they require. More than that, developers need to know how to apply guidelines in their workplace. Some standards must be followed strictly, whereas internal recommendations can be ignored if there are good reasons to do so. Distinguishing good from bad reasons is often tacit knowledge.

Requirements engineering: A piece of code is useful only with respect to a customer or user. If the code does not meet the requirements and expectations of the customer, it is useless. Because there may be different user groups, there will be different sets of requirements and expectations. Knowledgeable developers need to know a lot about all of them. Even well-commented and technically well-structured code cannot compensate for missing or misunderstood requirements. Therefore, requirements analysts need to understand the real requirements before they can be fulfilled. This is an important task and a major effort. It requires knowledge and insight into the application domain; the ability to communicate with domain experts; and technical skills to map requirements to possible solutions. Without the last ability, unrealistic requirements are uncovered too late in the process.

Requirements engineering refers to all activities related to requirements as outlined in Fig. 2.10: elicitation of requirements in interviews, workshops, and workplace observations. Facilitating negotiation between so-called stakeholders is also part of the job. A stakeholder is any person or group that is potentially affected by the software. By this definition, not only users but also managers, administrators, and even workers who may lose their jobs due to the new software must be considered stakeholders. Obviously, a subset of all those stakeholders must be identified and involved in the process. This requires interviews and workshops to be prepared, moderated, and analyzed. Software engineers need to master moderator tasks. There are informal and formal techniques to elicit, visualize, and validate requirements in

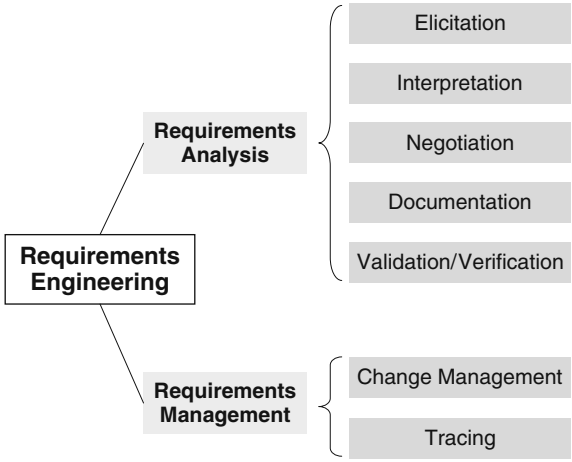

Fig. 2.10 Analysis and management activities in requirements engineering

those situations. When requirements surface, they need to be interpreted (to avoid misunderstandings and inconsistent use of terminology) and documented in a specification, set of use cases, or other forms. Of course, final specifications should be checked by the stakeholders to remove errors. During a project, requirements often change and need to be traced into design decisions. Requirements engineering is a subdiscipline of software engineering.

The gap between requirements engineering and software construction is bridged by software design. Rough decisions and structures are defined in the software architecture. This structure is then refined and filled during detailed design. As a result, specified requirements should be met by the constructed software, including associated documents. Functional requirements describe the features and functions of the program, whereas nonfunctional requirements refer to speed, volume of data processed, and other aspects. Architectural considerations need to take nonfunctional requirements into account: A suitable structure, use of frameworks, and distribution of components is often a prerequisite to reaching desired performance, maintainability, and flexibility.

Software engineers need technical **knowledge** in all areas affected by requirements, design, and software construction. In addition, they need experience in order to choose alternatives, prioritize and select stakeholders, and make informed decisions. For many of those decisions, there is no single optimal solution that could be taken from a textbook. Instead, experience and tacit knowledge must guide developers as decision makers. Software engineers make many decisions under uncertainty. Having access to more knowledge sources and experiences can increase confidence in their decisions.

2.4.1.3 Software Quality and Support

The above-mentioned basic activities are obviously not as basic as they first seemed. At first glance, they seem sufficient for developing software. However, high-quality software requires many supportive tasks to be carried out.

Most nontrivial software projects include more than one developer and several versions of code and documents. It is easy to lose track in that situation, in particular, if the workforce is geographically distributed. When contributions are integrated manually, the same modules may be modified by more than one person, leaving the system in an inconsistent or undefined state. Configuration management tools are the usual way of solving this problem. They offer a repository of artifacts, such as code modules of different size, and document chapters. A system is composed of a defined set of artifacts. The tool ensures conflicts will be detected and updated versions of all artifacts will be composed into a product release. Configuration management works in distributed settings and allows different variants of flexibility: "Optimistic locking strategies" allow multiple developers to access the same document d, as in Fig. 2.11. Developers add new documents and commit them. Others check out and may modify documents in parallel. Then, the first developer commits the changed version back to the system. If there are conflicting changes, they are detected when the second author tries to return his work. In that case, he needs to update his working copy, fix inconsistencies, and recommit. "Pessimistic locking" avoids this effect by locking all artifacts when someone checks them out for changing them. As a result, the above-mentioned conflicts cannot occur. At the same time, potential parallelism of work is rather limited.

Global software projects impose additional challenges, resulting from different time zones to cultural differences.

Nonfunctional requirements often refer to quality aspects like performance, maintainability, robustness, and so on. Assuring, maintaining, and managing software quality is yet another demanding (and knowledge-demanding) task within software engineering. Quality engineers or quality agents are a subset of software

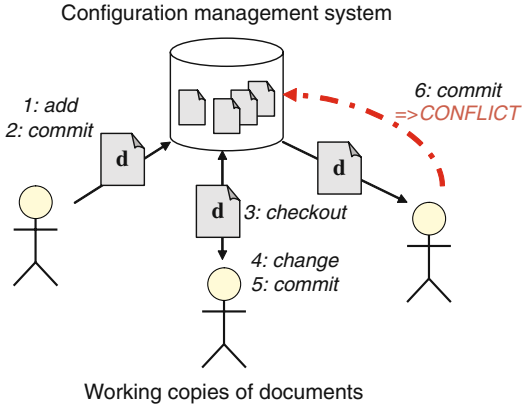

Fig. 2.11 Principle of optimistic locking in configuration management systems

engineers. Their tasks include checking code for errors. However, good quality must start much earlier in the development of software. Almost like a shadow project, software quality must be planned and pursued along the entire duration of the software project. Producing good quality requires formal and informal verification of development results. It should also include validation of requirements: Does the product reflect the customer requirements correctly? Did the requirements change since they were elicited?

Testing is a well-known subtask of quality assurance. In testing, a piece of code is executed with the goal of finding errors [75]. Of course, the final goal is to remove those errors and improve functional correctness and quality of the code. However, testing is almost a discipline in itself. If we want to uncover errors that produce wrong results or reactions, we need to know what the correct results are. Only if a test engineer (yet another software engineer!) prepares for testing, she creates a long list of test cases. A test case consists of the stimulus and the desired reaction or result. If we use many tests, we need many results. Unfortunately, desired results can only come from the customer or the specification. Who else could know what the customer desires? There are only very, very few cases in which desired results can be derived automatically from another formal source. In most cases, creating test cases includes a substantial amount of manual work (Fig. 2.12).

Knowledge helps to reduce effort in testing. There are several strategies to optimize the set of test cases. Ideally, a small set of test cases will find many errors. According to one test strategy, testers use only the specification to create "black-box test cases." Because the specification is considered a valid representation of the customer's desires, covering each requirement in the specification by at least one test case is a good heuristic. In "glass-box testing," testers look *into* the piece of code under test. They see more or less complex structures and data types. In glass-box testing, testers try to stimulate complex parts more frequently. They argue that errors are more likely to occur there. Knowledgeable testers use more sophisticated strategies and tools to determine the "coverage" of their test cases. When both all requirements and all structures in the code are stimulated ("covered") by test cases, testers can be quite confident to do a good job. However, complete testing of all possible executions of a program is infeasible in the general case, so testers have to resort to a strategy that has worked in the past. They know from experience that this strategy finds many defects. Software quality in general is an area that requires experience and a sense of economical pragmatism. For example, it may be desirable to review

ID	Set up	Parameters	Correct result
1	Set to 10:00 a.m.	1:45	11:45 a.m.
2	Set to 11:00 a.m.	2:15	1:15 p.m.
…			

Fig. 2.12 Two example test cases for a method that adds times and durations. Many more test cases are needed in a realistic testing environment

all documents produced. This implies a group of reviewers should read, comment, and discuss their findings. Reviews have been found to be very effective; up to 60% of all errors can be detected [38, 48]. At the same time, reviews are rather slow and take a lot of effort. Finding an adequate compromise requires knowledge of different variants of reviews and inspections with their properties [101]. More than that, quality agents need experience in planning and carrying out reviews effectively in their environment.

Management: Software engineers elicit requirements; they develop architectures and design code. Software engineers take care of software quality; they review documents and test code, among many other activities.

After a successful job as a programmer, a software engineer may be promoted to project leader. Managing projects is an important activity for software engineers – and it depends on knowledge and engineering. A project manager has many tasks. Project planning is the responsibility of a project manager. Planning depends on a good understanding of the required deliverables and a realistic estimation of effort and time to complete those deliverables. A knowledgeable project leader does not just "guess" effort and time. There are techniques to come up with sound estimations. They assume the project leader can assess and classify the project. Despite the formulas and techniques, most software estimations contain a good part of gut feeling and experience. Project managers provide work-breakdown structures, provide PERT charts, and milestone plans [53]. They control progress and consider error and quality measurements. And they present their projects to higher management and the customer. Of course, software engineers acting as project managers are also responsible for their fellow software engineers working in the project.

Software engineering is supposed to adopt a systematic and disciplined way of building software. In such an approach, successes should be repeatable. A learning organization will try to eliminate errors in their activities and improve their processes by building on proven practices. A process *prescribes* sequences and alternatives of activities to be carried out. Many processes also define roles and responsibilities. Deliverables are specified with respect to their roles in a process. Project managers have to instantiate and follow a project. On the one hand, a defined process will help the project manager and all participants to comply with good practices. On the other hand, an overly demanding process can put the project at risk. There are many examples of projects that made wrong decisions: Some neglect the process and deviated from plans and deliverables. They may finish in time but compromise quality and process conformance. Others strictly follow the process but are not able to complete the project in time. A project manager needs substantial experience and background to make a responsible decision for a good compromise. Processes incorporate past knowledge, but because technology and project pressures change constantly, there must be constant adaptation and tailoring.

Software projects involve risks. An interesting project cannot be sure to achieve all of its goals. Many things can go wrong: The customer may change her mind, making a lot of work obsolete. Developers may leave the company, removing essential knowledge from the team, which can delay progress. Or a subcontractor may not deliver in time, which can affect the project's own schedule. A risk is defined

as a potential problem that may occur but is not certain to occur. If it will certainly occur, we just call it a "problem," not a risk. A professional software process supports managers by risk management. **Risk management** is a technique for systematic handling of risks. Using checklists and iterative risk management procedures, risk managers identify risks through interaction with project participants. A risk is classified according to its probability and the potential damage it causes. The team will then develop mitigation plans for each of the top risks. Members of the team or the project leader are assigned mitigation tasks. In the above-mentioned example, a customer changed her mind about a requirement. A possible mitigation would be a contract that defines a price for each change request. A different mitigation can be an iterative process, which repeats requirements and construction in order to identify and fulfill changing requirements (Fig. 2.13).

Knowledge of many kinds is needed to lead a software project well. Of course, a project leader needs to know as much as possible about the problem to solve. A good estimation requires a realistic evaluation of environmental parameters. A good estimation is the prerequisite of a realistic project plan. All information available about the customer and his goals should be taken into account to reduce uncertainty and risk. Risk checklists from previous projects can provide a good start. Warnings and recommendations from the experience of other projects is an invaluable source for good decisions and compromises.

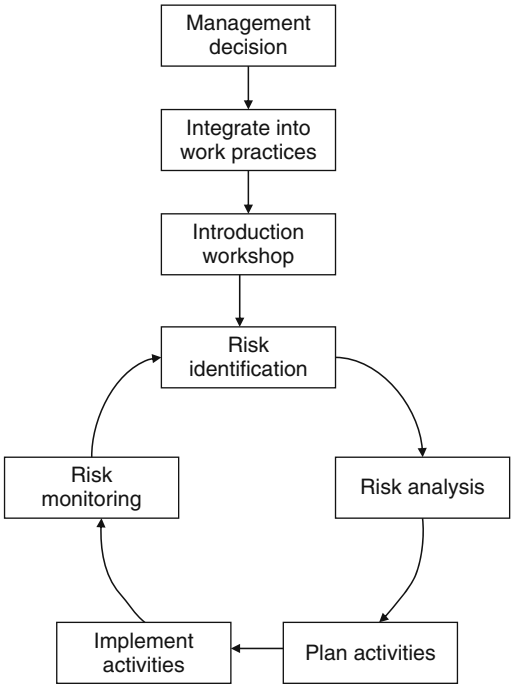

Fig. 2.13 Typical risk management process with set-up activities and iterative part

2.4.1.4 Domain Knowledge and Other Knowledge Areas

Software supports customers in carrying out their tasks. If you want to build good software, you need to know the customers and their tasks. For example, embedded software will run within a system or device constructed by electrical engineers. When requirements analysts talk to electrical engineers, they will face a specific attitude and language. When they talk to medical doctors about that same device, their perspectives and terminology will be very different.

Knowing the application domain with its traditions, terminology, and particularities is essential for success. Some domains, like safety-critical systems, have their own standards and rules. They are common to everyone working in the domain. Software engineers changing their working domain run the risk of not knowing some of the relevant standards. Ignoring them is a major risk, as it will lead to unusable software.

There are many other areas in software engineering with a huge number of additional knowledge areas. Agile methods, for example, offer a new approach to fast and flexible software projects [15, 16]. They apply incremental and iterative processes that involve the customer on a regular basis. Agile methods sound good in textbooks, but they require a lot of discipline and experience. Boehm and Turner [21], for example, claim that only those software engineers should work on agile projects who have proved their ability to work successfully in a traditional project.

Generating code is another technology that attracts a lot of attention [72]. The idea is old, but there are new contributions every now and then. In principle, code generation tries to produce more code faster than any programmer can write. The goal is to multiply development efficiency. Code is generated from macros or from models. If it is easier to draw a UML model than to write the code it represents, generating this code from the model is an appealing idea. However, using the code generators in an appropriate way requires a lot of knowledge. Should a certain piece of code be generated or written manually? Again, experience is needed to make informed decisions.

2.4.1.5 Summary

Obviously, basic textbook knowledge is rarely the problem in software engineering. Developers, quality personnel, and project leaders need good qualifications and a lot of technical expertise. In particular, decisions under uncertainty on several levels call for experience and specific knowledge.

Mnemonic 2.5 (Informed decisions under uncertainty)
In software engineering, experience and knowledge is needed to make compromises and decisions under uncertainty. There is no way of knowing the optimal decision. Context specifics and experiences must be taken into account.

The research of Orasanu and Connolly [81] into decision-making in practice showed that real-life settings include many of the following characteristics that are also typical of software engineering work. The following list applies their findings to the above-mentioned situation in software projects:

- Problems are ill-structured: It is difficult to see all relationships and dependencies of requirements, constraints, and stakeholders.
- Information is incomplete, ambiguous, or changing: Requirements change in all major software projects. Requirements tend to change because many stakeholders did not have sufficient expertise, imagination, or simply time to produce a concise and consistent set of requirements. Change is a symptom of growing understanding, which is one reason why agile software methods "embrace change" [15].
- Many participants contribute to the decisions, and goals are shifting, ill-defined, or competing: Stakeholders of a complex software project may include users, managers, and those workers who will lose their jobs when the software is installed. Usability goals compete with cost constraints and vague goals of saving jobs.
- Typically, time constraints exist and stakes are high, which makes decisions difficult and urgent at the same time. Limited information and knowledge access delimits well-informed decision making.
- The decision maker must balance personal choice with organizational norms and goals [81]. Software quality standards may be high – according to the company process model. An experienced quality engineer may know where corners can be cut e.g., in the case of a budget cut. Such a situation is often not explained in process manuals.

This short section on software engineering tasks and challenges has outlined the wide range of knowledge and experience needed in practice. It evades all phases and activities of a software project, and it ranges from simple factual information (on context, domain, requirement constraints, etc.) to delicate experiences and tacit assessment capabilities for decision making.

2.4.2 The Software Engineering Body of Knowledge

As the previous section shows, the range of knowledge in software engineering is huge. There is no exhaustive list of knowledge areas or knowledge transfer mechanisms. However, there is a reference classification of software engineering knowledge, the Software Engineering Body of Knowledge (SWEBOK) [56]. **SWEBOK** in itself can be regarded as a knowledge transfer mechanism: Every practitioner and software engineering scientist may have certain requirements in mind: What does a qualified software engineer need to know? SWEBOK is more than an intersection and less than a union of all those expectations. For our purpose of experience and knowledge management in software engineering, SWEBOK can serve different purposes:

- *Classification*: As we will see in Chap. 3, a stable, agreed-upon classification of terms is important for managing knowledge. Because of its broad approach, SWEBOK offers a category for most pieces of knowledge one can imagine in software engineering.

- *Checklist for personal development*: An individual software engineer may use SWEBOK as a reference to identify weak spots in his or her competencies. Individual learning can aim at closing any gaps. However, it is overly ambitious to aim for full coverage: Without practical project experience, many lessons can be learned only superficially. When working in a company, however, there is no need for a wish list like SWEBOK to identify room for improvement. Several deficits will become apparent in daily work. But there will be little time and opportunity to study what is missing.
- *Matching problems with existing knowledge*: This leads to the home ground of knowledge management: What can we do in a concrete, specific situation to act competently, although some knowledge is missing? SWEBOK can be used as a classification and index. When someone has a problem or demand, a common vocabulary will help to make the match with experiences and knowledge available in the organization.

Definition 1.4 (knowledge management) refers to the purpose of "solving a problem" as opposed to "completing computer science education." In general, knowledge management should be directed by actual or foreseeable demands, not personal interests in fictitious applications. Learning on demand is preferential in a company setting. This is where knowledge management takes place.

When we now look at SWEBOK, we do that with the above-mentioned considerations in mind: We are not trying to describe the perfect software engineer, but we see whether SWEBOK can help in classifying and indexing pieces of knowledge. At the same time, we treat SWEBOK as an attempt to structure a complex field of knowledge. This is a good exercise for Chap. 3, where structuring domain vocabulary is the focus.

Fig. 2.14 shows the outline of SWEBOK [56]. There are 10 main knowledge areas defined in SWEBOK. They are depicted as horizontal bars with attached subtopics. Each bar shows the knowledge area according to SWEBOK on the right. Related areas are labeled in SWEBOK as an additional knowledge area number 11. SWEBOK is represented by a rectangle that covers most of these knowledge area bars.

There is one more box in Fig. 2.14: it represents the growing number of related aspects and knowledge areas. They are considered outside software engineering (hence outside the SWEBOK rectangle) but are highly relevant for a knowledge management initiative in software engineering. Knowledge areas are grouped in Fig. 2.14 for the sake of this overview. Those groups called "core activities," "quality and support," and "management" were introduced in the previous section. They show where those discussions feed into SWEBOK.

There are recurring patterns of subtopics in the knowledge areas: Concepts or Basic Concepts introduce the knowledge area and its terminology. There are Key Issues to detail certain aspects. In some fields, there is an established set of activities, such as in Software Requirements: the labels are almost identical to the ones used in Fig. 2.10 and in many other sources on requirements engineering. Other labels reflect the challenge of finding useful abstractions, for example "Techniques

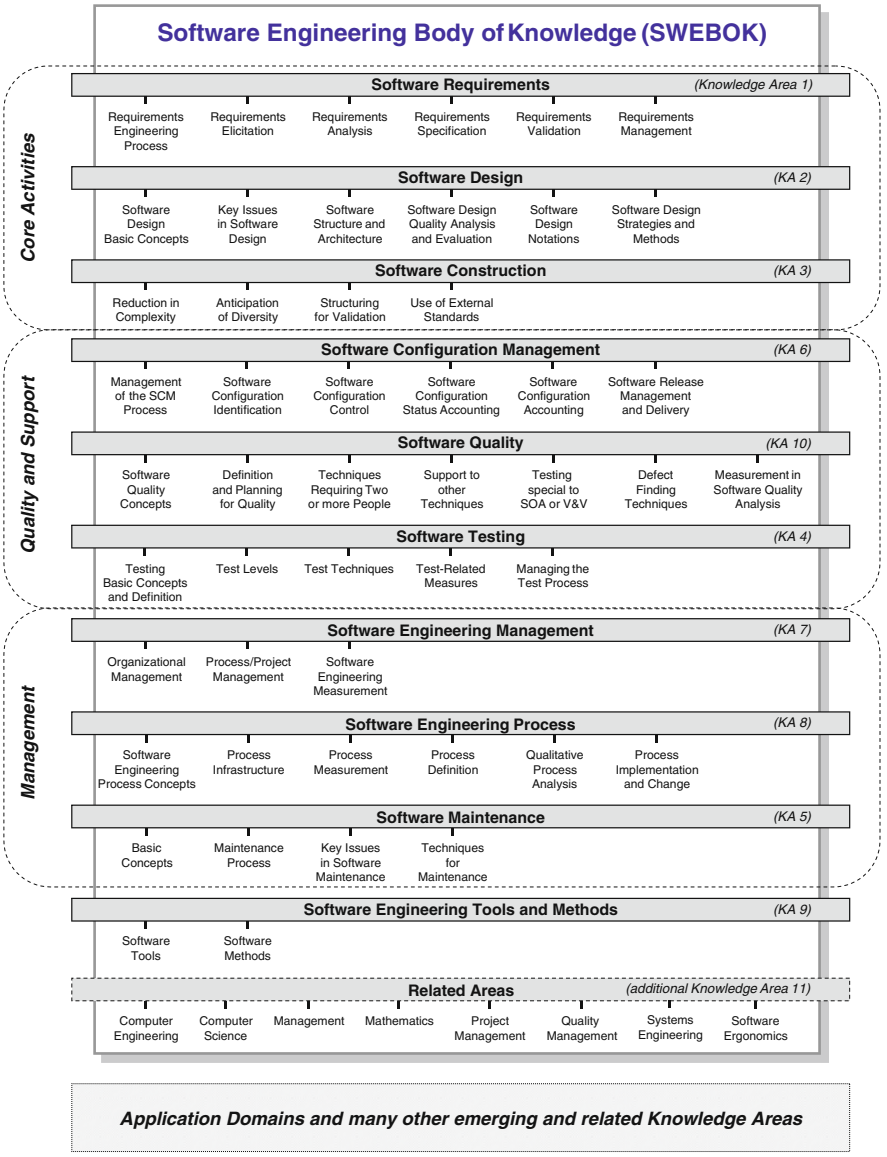

Fig. 2.14 SWEBOK structure as a classification scheme for matchmaking [56]

requiring two or more people" within Software Quality. It is difficult to summarize and classify all the knowledge relevant for an emerging and quickly growing field like software engineering. Not all new trends can be covered by SWEBOK, but a big portion of the core of software engineering is captured and organized. Fig. 2.14 could be further detailed to show the concrete types of tools, methods, and test

aspects that are covered by SWEBOK. Some important tasks like risk management are not depicted on this level of abstraction, but they are definitely a crucial piece of knowledge in software engineering. Most other aspects presented during the short introduction to software engineering tasks in the previous chapter can be located easily within the SWEBOK outline.

SWEBOK is proposed as an agreed-upon structure of the discipline of software engineering. Using SWEBOK as a semantic structure can help software engineers and knowledge managers alike to come to a better mutual understanding. In very concrete terms, a defined outline like SWEBOK facilitates searching beyond keyword search. It will also facilitate communication in the software engineering community.

The authors of SWEBOK made a clear decision about the scope of this body of knowledge. Domain knowledge is not considered part of software engineering knowledge:

> Software engineers must not only be knowledgeable in what is specific to their discipline, but they also, of course, have to know a lot more. The goal of this initiative is not, however, to inventory everything that software engineers should know, but to identify what forms the core of software engineering. It is the responsibility of other organizations and initiatives involved in the licensing and certification of professionals and the development of accreditation criteria and curricula to define what a software engineer must know outside software engineering. We believe that a very clear distinction must be made between the software engineering body of knowledge and the contents of software engineering curricula. (www.swebok.org)

This distinction is useful for framing software engineering terminology and context. There are other experts in application domains like medicine or automotive who may develop their own body of knowledge. Therefore, application domain knowledge is considered outside the SWEBOK. It is not drawn within the boundaries of SWEBOK according to Fig. 2.14. However since it is closely related to the software engineering knowledge, it is depicted in an extra gray rectangle just below SWEBOK. It is part of the knowledge software engineers have to master in practice. It does not matter who structures and publishes those related bodies of knowledge.

2.4.2.1 Evaluation of SWEBOK as a Matchmaker

How appropriate is SWEBOK as a matchmaker? Imagine a situation in which you need a piece of information or knowledge: Where would you look it up in SWEBOK? If you can easily identify two or three subsections to look into, SWEBOK is a powerful classifier. It guides you to a small number of places to check. If, however, no label really fits or if too many categories might be relevant, SWEBOK is less adequate.

But even if you are sure where to look: Will relevant material really be there?

From the knowledge authors' point of view, the situation looks different: Without knowing future demands and problems that could possibly be addressed by a piece of knowledge they want to store, authors are asked to categorize it. This is more difficult than one might think. Any real problem or solution will touch on more than one category. Should an experience be classified by problem area, or by solution, or by some blend of both? How should we consider context? It is often a hard piece of

work to imagine what future users might want to know. *Engineering* knowledge and experiences is concerned with that question, among others. Engineering includes structuring, indexing, and comparing with other knowledge. Chapter 3 is devoted to engineering knowledge for reuse.

SWEBOK refers mainly to "hard software knowledge": technical skills, not soft skills. Many problems in software projects go back to personal conflicts, misunderstandings, and poor social skills. When we look at software engineering knowledge from that angle, we definitely need to add corresponding categories of knowledge.

The discussion about SWEBOK as a matchmaker shows:

- It is not easy to develop a good classification scheme for experiences and knowledge. An index looks different from the authors' and the users' points of view. Effective matching requires engineering of knowledge.
- Matching keywords will rarely do: Each piece or demand might need several keywords or categories to describe it.
- SWEBOK has many merits that we did not even mention here. We are only interested in knowledge management for the software engineering area. Other goals and contributions have been omitted.

For the remainder of this book, we treat SWEBOK as a good overview of knowledge areas within software engineering. We are aware that it is not the one and only possible classification scheme for software engineering. As we will see later, part of a classification should grow out of the problems and applications – not be imposed top-down. In essence, there is no static borderline around software engineering knowledge relevant for knowledge management.

2.5 Appropriate Knowledge Management for Software Engineering

Before Chap. 3 starts the discussion of techniques, we should mention the diversity in knowledge management research. There is not just one orientation but a whole variety of research directions.

In a workshop on learning software organizations and requirements engineering (LSO+RE 2006), the editorial discusses what makes software engineering special as a domain for knowledge management [20].

Software process improvement intersects with learning organizations. Both fields aim at improving efficiency; both fields apply iterative approaches to feedback and learning. The concept of "Experience Factory" [11] is the first well-known systematic approach to organizational learning in the software engineering field [11, 31]. It will be discussed in detail in Chap. 4.

What sets a learning *software* organization apart from other learning organizations? Software development is a very knowledge-intensive form of work. Software organizations are also more mature in the usage of information technology. In fact, the input and outcome of software engineering is information. As a consequence, we might expect software organizations to make better use of available tools.

Table 2.1 Schools of knowledge management, according to Earl [35]

			Attribute			
			Focus	**Aim**	**Unit**	**Philosophy**
School	**Technocratic**	Systems	Technology	Knowledge bases	Domain	Codification
		Cartographic	Maps	Knowledge directories	Enterprise	Connectivity
		Engineering	Processes	Knowledge flows	Activity	Capability
	Economic	Commercial	Income	Knowledge assets	Know-how	Commercialization
	Behavioral	Organizations	Networks	Knowledge pooling	Communities	Collaboration
		Spatial	Space	Knowledge exchange	Place	Connectivity
		Strategic	Mindset	Knowledge capabilities	Business	Consciousness

Earl [35] has developed a framework to categorize studies on knowledge management according to different research directions, which he calls "schools," as shown in Table 2.1. The "technocratic" approach to knowledge management focuses on systems, cartography (maps), and engineering of knowledge, whereas the "economic" school looks at the commercial value of knowledge. "Behavioral" approaches focus on organizational, spatial, and strategic aspects of knowledge management.

As Table 2.1 shows, each direction focuses on different aspects that are rooted in a specific philosophy and attitude toward knowledge management. There is a typical aim and associated focus and a certain kind of unit that will often pursue the respective kind of knowledge management aspect.

Example 2.6 (Systems school)

For example, a philosophy of codification ("knowledge needs to be coded explicitly") may aim at building knowledge bases. For those repositories, technology is an obvious focus. Often, an entire domain of knowledge is addressed. For example, building a knowledge base for estimating software project duration and cost may be pursued by setting up a data exchange system for the estimation community. CeBASE is a software engineering initiative for developing experience bases in a community [76].

Example 2.7 (Organizational school)

Someone convinced of the power of collaboration (as a philosophy) may aim at pooling knowledge, including tacit knowledge. For a community, building a network may be the focus. For example, a department head may decide to institutionalize the exchange of software quality experts (a community) by inviting them to regular meetings. Networking can also be supported by yellow pages or other systems, but such a technocratic approach would rather point to a cartographic view. Usually, an initiative reflects a certain mixture of different influences.

We will use Table 2.1 as another map for orientation. Real companies will need to exploit several knowledge management aspects together. In the following chapters, techniques can be mapped to aspects to find similarities. Maps or mental models

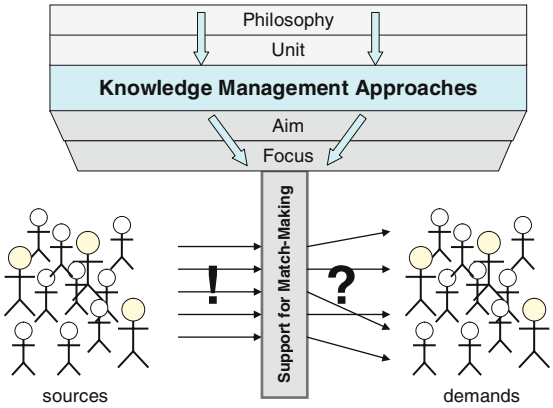

Fig. 2.15 Philosophy and kind of unit point to appropriate support for matchmaking

help people (and organizations!) to develop shared understanding. As a reference for problems, existing experiences, and ongoing research, they can be catalysts. Whereas SWEBOK was a rather linear reference for software engineering knowledge, the schema provided by Earl's table provides categories for knowledge management approaches. As Fig. 2.15 illustrates, knowledge management approaches can be characterized by the philosophy they represent and by their respective unit. Those search criteria lead to an approach, which implies an aim and a focus to pursue. The selection will be an approach that complies with the philosophy and kind of unit. All selected approaches support making the match between those in demand of knowledge and those people who have that kind of knowledge – tacitly or explicitly.

There are some general lessons learned associated with picking knowledge management approaches. They can be used as general guidelines for designing appropriate knowledge management initiatives:

- Managing knowledge is more successful if there is already something to manage. Starting without sources and without existing knowledge is difficult. Imagine you need some information on a software tool. If software engineers find an empty knowledge base, and no links to experts, this would not help a lot. Therefore, knowledge management in software engineering must provide an initial content for a base before it is delivered to its users. This is called a "seed."
- Capturing knowledge without concrete demands is difficult, too. The purpose of knowledge management is not to store and encode knowledge, but to deliver it to those who need it. Knowledge acquisition needs to be aware of what is needed. Imagine someone who has been working with a tool for a while. How should this person know whether it is worth the time to put something into the knowledge base about that tool? And what kind of information would be most appropriate?
- There is good potential for reuse if many knowledge workers need the same kind of knowledge frequently. Imagine the company using a software tool only for certain tasks (e.g., risk management). Risk management should be carried out by each project, but only as a background activity. As a consequence, knowledge about the tool is needed on a regular basis, but not frequently enough for everyone to know it by heart.

- Knowledge that can be acquired or internalized fast is more likely to make knowledge management a success. If learning takes very long or requires a teacher and exercises, that type of knowledge is less adequate for knowledge management. It should rather be learned in a formal education program. In the example of the risk management tool, gaining basic awareness about risks in software projects is a slow and tedious topic. But if someone knows those basics already and finds some hints and checklists to make risk management more effective, this is a faster and more promising approach.
- The **granularity** of meaningful pieces of knowledge should be small to medium. Very small chunks, like the meaning of an acronym, can be handled with nonspecific search engines and hardly justify expensive knowledge initiatives. However, very voluminous packages of knowledge take too much time and are difficult to evaluate for relevancy (see item above). Medium granularity provides substantial support but is still economical to read, evaluate, and internalize when needed.
- **Tacit knowledge** is important and must not be excluded. Restricting an initiative to explicit knowledge cuts out the source of most interesting experiences and may restrict it to single-loop learning [7, 8].
- However, relying on tacit knowledge alone is often too time-consuming and takes too much effort. Knowledge management techniques can unfold their capabilities best when there is also a reasonable portion of explicit knowledge to spread.

Ideally, knowledge workers already know a lot about the software engineering domain they are working in. Knowledge management can build on this prior knowledge and enable knowledge workers to exchange details, facts, new rules, and hints. New knowledge combines with existing knowledge, as Definition 1.7 (knowledge) implies. Tacit and explicit sources and mechanisms for making matches can be exploited. In short, knowledge management is most promising in software engineering when all parts of Fig. 1.8 are activated.

For example, software engineers in an automotive company know how to build brake system software. They benefit from a knowledge management approach that does not need to convey all their basic knowledge but supports them with details of brake hardware, new safety regulations, and experiences from their fellow software engineers. It should also support them in feeding back what they have learned individually. There are both experience exchange opportunities and mechanisms for capturing and externalizing tacit knowledge. Newly created knowledge will be engineered and matched to future demands. An appropriate approach can be ambitious. It should be neither oversized nor too modest. Appropriate approaches meet the above-mentioned success criteria.

Example 2.8 (Supporting managers)

Supporting software managers is a promising field within software engineering. Managers need data and information to plan a project. They need progress information to control it. And they need support from simulations (based on "explicit mental simulation models") and experiences for predicting and estimating project parameters. There are many chunks of medium-sized information. Several pieces

of knowledge will be communicated from person to person because no one writes down all his gut feelings. Managers will still make their decisions by themselves: Simon [102] showed that managers do not act fully rationally; the bounded rationality they apply is informed by adequate knowledge management, but there is no need to oversize heuristic support. The decision *will and should* be made by the managers, not by a resolution mechanism.

2.6 Types of Tools Supporting Knowledge Management

Earl's classification of knowledge management schools illustrates the broad variety of approaches, goals, and mechanisms for supporting knowledge management. Some are tightly associated with a certain kind of tool like knowledge directories or bases. Some specific tools will be mentioned in the respective chapters below (e.g., for ontologies and mind maps).

Many aspects of knowledge management may be carried out manually, but a larger number of knowledge workers usually require tools, too. For that reason, tools are fundamental parts of knowledge management. Even without going into detail about any particular tool, there are two types of tools that are associated with their respective type of knowledge management approach.

Our map of knowledge management (Fig. 1.8, repeated below as Fig. 2.16) shows many tasks in perspective, starting with the identification of appropriate sources of

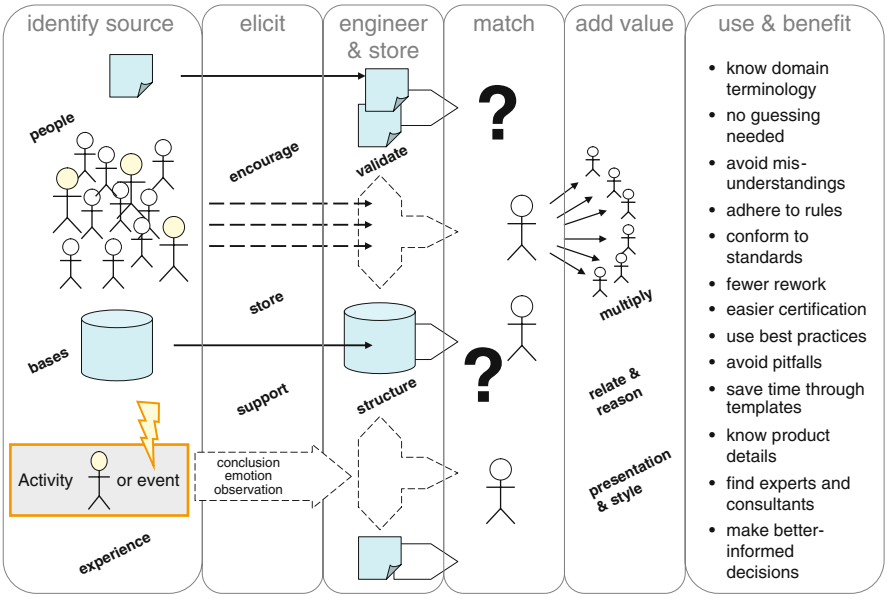

Fig. 2.16 Map (repeat of Fig. 1.8) to show product- and process-centered aspects

knowledge. Eliciting tacit and explicit knowledge leads to structuring, engineering, and storing knowledge. To add value, a match must be made between available material (sources) and knowledge workers who need this material.

If a single tool had to support all those aspects, it would need to be highly specific and very versatile. Because all companies are different, tailoring would be important. That makes a tool very expensive. In practice, relying on existing components will be less expensive *and* more powerful. If a new tool is introduced, learning cost often exceeds tool license fees. In addition, the psychological barrier can be a problem. Knowledge workers must have good reasons to adopt yet another tool. They will not do it when they *perceive or believe* the new tool only replicates functionality of well-known tools they already know. Therefore, providing adequate information on new benefits must be part of the introduction.

Example 2.9 (Supporting communication)

Imagine communication about knowledge: Would you rather use your familiar e-mail tool, or would you prefer to learn using a dedicated tool with similar features? How would you like checking for messages in two tools? Since knowledge management is itself a discipline *supporting* software engineering, it should not erect unnecessary hurdles for mastering new tools. Instead, a knowledge management initiative should identify and bundle existing tools and relevant features and help to use them for exploiting knowledge resources better. Specific tools will mostly be limited to highly specific tasks (e.g., for automated reasoning).

These general remarks apply to almost all environments: Tools should be picked and combined according to the existing tool base in a company. There is a huge diversity of customary tools, and it is impossible to even list them all. Instead, we want to look at two families of tools that relate to different viewpoints on knowledge management: the product view and the process view. Both occur in our map, but both call for different tools.

Mentzas et al. [73] compare the two approaches nicely. The difference is deeply routed in different philosophies and schools: One of them views knowledge as a product (or a "thing" to be moved around). The other considers knowledge a process; as in Definition 1.7, knowledge is treated as something residing in people's minds.

Product-centric approaches to knowledge management are concerned with knowledge logistics: how to package and store and classify a "piece" of knowledge in order to find and "deliver" it when needed. In this world view, knowledge workers live in a supermarket of more or less "tasty" servings of knowledge. It is up to the supermarket management to organize the offers and ensure the freshness of the products. Knowledge workers are responsible for selecting products with the support of the structures and pointers created by management. They will swallow and digest those pieces at their own pace. When new chunks of knowledge come in from "somewhere," they are checked, repackaged, labeled, and put on the shelves. Artificial intelligence has pursued a similar approach in computer science. Approaches from that direction are often technocratic (cf. Earl [35]) and

product-centric. Powerful mechanisms are available in this field. Search and information retrieval tools help to match demands and offers. Using metadata helps to go far beyond keyword search. Pieces of knowledge are stored in expert systems and knowledge-based systems. Reasoning mechanisms use ontologies and Semantic Web technology for automated classification, comparison, and mining of information relevant to a task at hand [73]. Selected approaches are explained in more detail in Chap. 3.

Process-centric approaches to knowledge management are centered on the learning of individuals and teams. Because knowledge resides in people, networking is a social, psychological, and cognitive necessity. Empowering people to carry out learning loops and reflection is the focus. In this world view, knowledge workers live in a community of chefs in a gourmet restaurant. Ingredients for their great meals need to be fresh, and that is a concern. However, much more important is the tacit experience that their colleagues have acquired when working in various great restaurants before.

Example 2.10 (Socialization)

Kitchen apprentices watch the master chefs and copy what they see: Socialization is taken very seriously. New knowledge about recipes and refinements is created by bringing experts together in an empowering environment. Knowledge consumers out in the restaurant enjoy the sophisticated solution and benefit, too. A gourmet restaurant does not scale up to feed as many people as a supermarket.

Yellow pages help to establish and maintain networks. Technologies for exchanging information in a group, such as groupware, and computer-supported cooperative work will support group learning: They facilitate remote or multimodal meetings. People modify the same document while they talk or chat over the Internet.

Obviously, a product-centric view alone will miss a large portion of knowledge creation and engineering potential. On the other hand, a purely process-centric approach might not be sufficient to feed a large company. It will usually be a wise decision to go for a balance between both views. There is no principal competition between those world views – but proponents typically come from different research and work backgrounds. This might lead to a (tacit) controversy. You should rather pick the best of both worlds: Why not have great knowledge masters refine knowledge and disseminate it with product-centric logistics and formalisms?

2.7 Problems for Chapter 2

Problem 2.7.1: Single- and double-loop learning
What is the difference between single-loop and double-loop learning? Explain both modes and give a short example from software engineering for each of the two.

Problem 2.7.2: Framework XY learning scenario
A software company has used a certain framework XY for building business applications. Problems using that framework are reported to a hotline. The hotline releases a new version of its newsletter "XY Procedures." New projects are supposed to follow those procedures. This is supposed to spread learning through all projects. What kind of learning is this? How could this type of learning turn out to be counterproductive? Refer to the XY example. What concrete activities could help to prevent that negative effect?

Problem 2.7.3: Formal reasoning patterns
You are working as a test engineer. Over the years, you have noticed that many tests fail due to the incomplete initialization of variables. Describe the three formal reasoning patterns (abduction, deduction, induction) and label the following statements with the corresponding reasoning pattern name:

- *". . .there are often errors in initialization."*
- *"Initialization looks so trivial, so many people are not very careful about it."*
- *"Programmers make more and more mistakes."*
- *"Setting a counter to 0 or 1 is often forgotten."*
- *"Flawed initialization is easy to find by testing; that is why we find so much, because we test more systematically."*
- *"Programs have errors."*

Problem 2.7.4: Schools and approaches of knowledge management
A company wants to encourage the exchange of knowledge among its software engineers. For each of the following suggestions, identify the respective school according to Earl and approach (product- or process-centric) it belongs to:

- *"We develop a knowledge database and send an e-mail to all software engineers asking them to enter their knowledge."*
- *"Let's put an info terminal in the main lobby; every time you enter the building, you will see a new 'knowledge item of the day.'"*
- *"We could use a Wiki Web to record everything we think others might need."*
- *"Okay, but there needs to be a moderator; plus, we should have monthly meetings of the Wiki User Group."*
- *"Let's put up a coffee machine and two sofas."*
- *"There are powerful networks now, so we can even send movies. Everybody gets a camera on their desk, and when they want or have something interesting, they record a movie."*
- *"Great idea! We hire a person who can index all incoming movies according to their SWEBOK category, and a few other attributes. We build a resolution machine that helps to match new entries with stored ones."*

Problem 2.7.5: Knowledge management life-cycle
Draw Nonaka and Takeuchi's knowledge management life-cycle. Explain it by applying the concepts to the example of a group of software project leaders who meet in a community of practice (CoP) to learn about cost estimation.