Method Engineering as a Social Practice

3

Summary of What We Will Learn in Chapter 3

- The importance of social practices in SME
- The concept of method rationale
- How social practices are incorporated into evolutionary method engineering and method-user-centred method engineering

Formalised systems development methods are used in systems development as a means to express and communicate knowledge about the systems/software development process (Ågerfalk and Fitzgerald 2006). Since methods are social constructs, they embed various assumptions about people and systems development as a social practice (Introna and Whitley 1997; Russo and Stolterman 2000). Essentially, methods encapsulate knowledge of good design practice so that developers can be more effective, efficient and confident in their work. Nonetheless, it is a well-known fact that many software organisations do not use methods (Iivari and Maansaari 1998; Nandhakumar and Avison 1999) and, when methods are used, they are not used straight out of the box but are tailored to suit the particular development situation (Fitzgerald et al. 2003). This tension between the method 'as documented' and the method 'in use' has been described as a 'method usage tension' between 'method-in-concept' and 'method-in-action' (Lings and Lundell 2004).

If a method is to be accepted and used, method users must perceive it as useful in their development practice (Riemenschneider et al. 2002). In general, for someone to regard a piece of knowledge as valid and useful, it must be possible to rationalise that knowledge, i.e., it must make sense to developers and be possible to incorporate into their view of the world.[1] This is particularly true in the case of method

[1] Ethnomethodologists refer to this property of human behaviour as 'accountability' (Garfinkel 1967; Dourish 2001; Eriksén 2002); people require an account of the truth or usefulness of

B. Henderson-Sellers et al., *Situational Method Engineering*,
DOI 10.1007/978-3-642-41467-1_3, © Springer-Verlag Berlin Heidelberg 2014

prescriptions since method users are supposed to use these as a basis for future actions, and thus use the method description as a partial account of their own actions. Hence, the type of knowledge that is codified as method descriptions can best be understood as a form of 'action knowledge' (Goldkuhl 1999; Ågerfalk et al. 2006).

In order to understand better the rationalisation of system development methods, several different approaches have been investigated. Each examines the pros and cons of different alternatives and the impact of making specific choices. One approach is that of eliciting the requirements for the methodology using typical strategies from the requirements engineering literature (as discussed further in Sect. 6.2), perhaps using a goal-based approach (see the overview of several such goal-oriented requirements engineering (GORE) approaches by Lapouchnian 2005, and also one particular example in Sect. 6.4.2).

In this chapter, however, we focus on the concept of method rationale as developed in the literature (Oinas-Kukkonen 1996; Ågerfalk and Åhlgren 1999; Ågerfalk and Wistrand 2003; Rossi et al. 2004; Ågerfalk and Fitzgerald 2006). Method rationale concerns the reasons and arguments behind method prescriptions, and why method users (e.g., systems developers) choose to follow or adapt a method in a particular way. This argumentative dimension is an important but often neglected aspect of systems development methods (Ågerfalk and Åhlgren 1999; Ågerfalk and Wistrand 2003; Rossi et al. 2004). One way of approaching method rationale is to think of it as an instance of 'design rationale' (MacLean et al. 1991) that concerns the design of methods, rather than the design of computer systems (Rossi et al. 2004). This aspect of method rationale captures how a method may evolve and what options are considered during the design process, together with the argumentation leading to the final design (Rossi et al. 2004), thus providing insights into the process dimension of method development. A complementary view on method rationale is based on the notion of purposeful-rational action. This aspect of method rationale focusses on the underlying goals and values that make people choose options rationally (Ågerfalk and Åhlgren 1999; Ågerfalk and Wistrand 2003). It also provides an understanding of the overarching conceptual structure of a method's underlying philosophy.

3.1 Methods as Action Knowledge

A method description is a linguistic entity and an instance of what can be referred to as action knowledge (Goldkuhl 1999; Ågerfalk 2004). The term 'action knowledge' refers to theories, strategies and methods that govern people's action in social

something in order to accept it as valid. According to ethnomethodologist Harold Garfinkel (1967), actions that are accountable are 'visibly-rational-and-reportable-for-all-practical-purposes'.

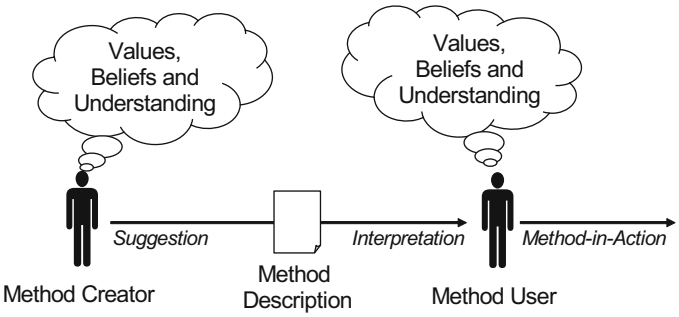

Fig. 3.1 Method descriptions in a communication context (after Ågerfalk and Fitzgerald 2006). Reprinted with permission of the publisher

practices (Goldkuhl 1999). The method description is a result of a social action[2] performed by the method creator directed towards intended users of the method. A method description should thus be understood as a suggestion by the method creator regarding how to perform a particular development task. This 'message' is received and interpreted by the method user, and acted upon by following or not following this suggestion (see Fig. 3.1), i.e., by transforming the method description (or 'formalised method') (Fitzgerald et al. 2002) or 'method-in-concept' (Lings and Lundell 2004) into a method-in-action. The 'method as message' is formulated based on the method creator's understanding of the development domain and on his or her fundamental values and beliefs. In such a team-based environment, shared understanding is critical—this may be implicit or explicit, some of which may be true and some false (Fig. 3.2). Similarly, the interpretation of a method by a method user is based on his or her understanding, beliefs and values.

It is possible to distinguish between five different aspects of action knowledge: a *subjective*, an *intersubjective*, a *linguistic*, an *action* and a *consequence* (Goldkuhl 1999; Ågerfalk 2004). Subjective knowledge is part of a human's 'subjective world' and is related to the notion of 'tacit knowledge' (Polanyi 1958). Subjective knowledge is shown as two 'clouds' in Fig. 3.1. This would be the type of knowledge that someone possesses after having interpreted and understood a method. Intersubjective knowledge is 'shared' by several people in the sense that they attach the same meaning to it and are able to meaningfully communicate (parts of) it among themselves. This could imply that the communicator (method creator) and interpreter (method user) agree on some of the elements of the 'clouds' in Fig. 3.1, and that they thus attach the same meaning to, at least parts of, a particular method. Linguistic knowledge is expressed as communicative signs, for example, as the written method description in Fig. 3.1. As the name suggests, action knowledge is expressed, or manifested, in action. This is the action aspect of knowledge

[2] According to sociologist Max Weber, social action is that human behaviour to which the actor attaches meaning and which takes into account the behaviour of others, and thereby is oriented in its course (Weber 1978).

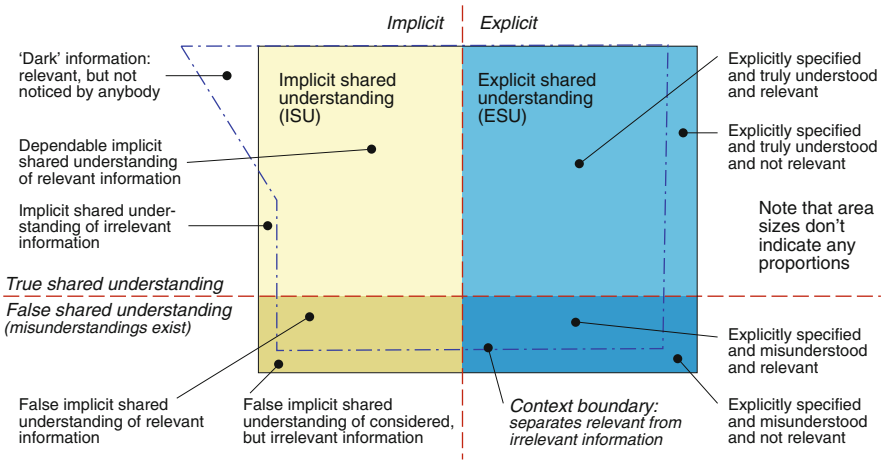

Fig. 3.2 Forms and categories of shared understanding: implicit versus explicit and true versus false. In addition, this diagram identifies the possibility of 'dark' information, i.e., information that no stakeholders are aware of (after Glinz and Fricker 2013)

or 'method-in-action'. Finally, traces of the action knowledge might be found in materialised artefacts, which constitute a consequence aspect of the knowledge. This would correspond to, for example, models and documentation produced as well as the actual software developed.

3.2 Method Stakeholders

When we think of software and systems development methods, what usually spring to mind are descriptions of ideal typical[3] software processes. Developers use such descriptions in practical situations to form what can be referred to as methods-in-action (Fitzgerald et al. 2002). Method engineering acknowledges that a method used in an actual project typically deviates significantly from the idealised process described in method handbooks and manuals (Iivari and Maansaari 1998; Nandhakumar and Avison 1999; Fitzgerald et al. 2003). Such adaptations of methods can be made more or less explicit and be based on more or less well-grounded decisions.

Methods need to be tailored to suit particular development situations (see also Chap. 7) since a method, as described in a method handbook, is a general

[3] Max Weber introduced the notion of an 'ideal type' as an analytic abstraction. Ideal types do not exist as such in real life, but are created so as to facilitate discussion. We use the term here to emphasise that a formalised method, expressed in a method description, never exists as such as a method-in-action. Rather, the method-in-action is an appropriation of an ideal typical formalised method to a particular context. At the same time, a formalised method is usually an ideal type created as an abstraction of existing 'good practice' (Ågerfalk and Åhlgren 1999).

description of an ideal process. Such an ideal type needs to be aligned with a number of situation-specific characteristics or 'contingency factors' (van Slooten and Hodes 1996; Karlsson and Ågerfalk 2004).

When a situational method has been devised, or 'engineered', and is used by developers in a practical situation, it is likely that different developers disagree with the method description and adapt the method further to suit their particular hands-on situational needs. As a consequence, the method-in-action will deviate not only from the ideal typical method but also from the situational method.

Altogether, this gives us three 'abstraction levels' of method: (a) the ideal typical method that abstracts details and addresses a generic problem space, (b) the situational method that takes project specifics into account and thus addresses a more concrete problem space and (c) the method-in-action, which is the manifestation of developers' actual behaviour 'following' the method in a concrete situation. It follows from this that both the ideal typical method (a) and the situational method (b) exist as linguistic expressions of knowledge about the software development process (middle 'level' of Fig. 1.8). At the same time, the method-in-action represents an action aspect of that knowledge, which may of course be reconstructed and documented post facto (in addition to the way it is manifested in different developed artefacts along the way) (lower 'level' of Fig. 1.8).

Figure 3.3 offers an alternative visualisation of these three abstraction levels of method and corresponding actions and communication between the actors involved. In Fig. 3.3, the Method User of Fig. 3.1 has been specialised into the Method Configurator and the Developer (method creators and method configurators are collectively referred to as method engineers). Method configurators use the externalised knowledge expressed by the method creator in the ideal typical method as one basis for method configuration and subsequently communicate a situational method to developers. What is not shown in Fig. 3.3 is that method construction, method configuration and method-in-action rely on the actors' interpretation of and assumptions about the development context. The developer 'lives' within this context and thus focusses his or her tailoring efforts on a specific problem space. The method creator, on the other hand, has to rely on an abstraction of an assumed development context and thus focusses on a generic problem space. Finally, the method configurator supposedly has some interaction with the actual development context, which provides a more concrete basis for configuring a situational method.

In both method construction and method configuration, the method communicated is a result of social action aimed towards other actors as a basis for their subsequent actions. This means that method adaptation, in construction, configuration and in-action, relies on the values, beliefs and understanding of the different actors involved—and this is where method rationale comes into play.

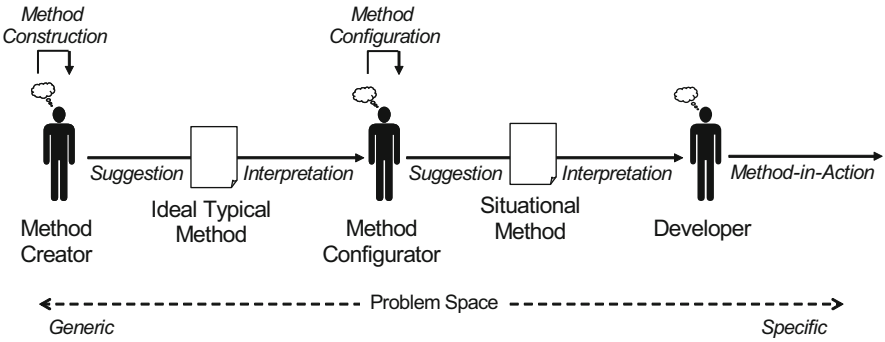

Fig. 3.3 Levels of method abstraction in methods as action knowledge (after Ågerfalk and Fitzgerald 2006). Reprinted with permission of the publisher

3.3 Method Rationale

Since methods represent knowledge, they also represent rationale. Therefore, a method user 'inherits' both the knowledge expressed by the method and the rationale of the method constructor (Ågerfalk and Åhlgren 1999). It can be argued that, regardless of the grounds, method tailoring (both during configuration and in-action) is rational from the point-of-view of the method user (Parnas and Clements 1986) who must decide whether to follow, adapt or omit a certain method or part thereof. Such adaptations are driven by the possibility of reaching 'rationality resonance' between the method and the method users (Stolterman and Russo 1997). That is, they are based on method users' efforts to understand and ultimately internalise the rationale expressed by a method description.

From a process perspective, method rationale can be thought of as having to do with the choices one makes in a process of design (Rossi et al. 2004). Thus, we can capture this kind of method rationale by paying attention to the questions or problematic situations that arise during method construction. For each question, we may find one or more options, i.e., 'solutions' to that question.

As an example, consider the construction of a method for analysing business processes. In order to graphically represent flows of activities in business processes, we may consider the option of modelling flows as links between activities, as in UML Activity Diagrams (OMG 2010). Another option would be to use a modelling language that allows for explicitly showing communicative and material results of each action and how those results are used as a basis for subsequent actions, as in VIBA[4] Action Diagrams (Ågerfalk and Goldkuhl 2001; Ågerfalk 2004). To help explore the pros and cons of each option, we may specify a number of criteria as guiding principles. Then, for each of the options, we can assess whether it

[4] Versatile Information and Business Analysis.

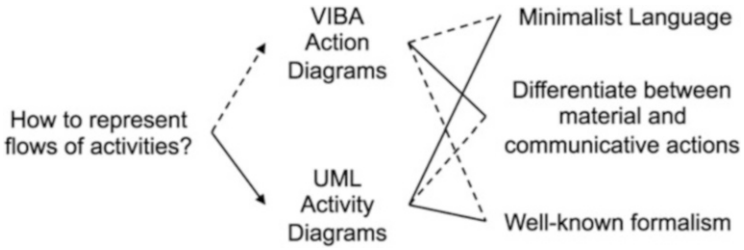

Fig. 3.4 Method rationale as choosing between options VIBA Action Diagrams and UML Activity Diagrams for modelling activity flows (based on the Question, Option, Criteria Model of Design Space Analysis (MacLean et al. 1991)). The solid arrow between 'situation' and 'option' indicates the preferred choice; a solid line between an option and a criterion indicates a positive impact, while a dashed line indicates a negative impact (after Ågerfalk and Fitzgerald 2006). Reprinted with permission of the publisher

contributes positively or negatively with respect to each criterion. Let us, for example, assume that one criterion (a) is that we want to create a visual modelling language (notation) with as few elements as possible in order to simplify models (a minimalist language). Another criterion (b) might be that we want a process model that is explicit regarding the difference between material actions and communicative actions[5] in order to focus developers' attention on social/communicative aspects and material/instrumental aspects, respectively (thus a more expressive language). Finally, a third criterion (c) might be that we would favour a well-known modelling formalism. The UML Activity Diagram option would have a positive impact on criteria a and c, and a negative impact on criterion b, while the VIBA Activity Diagram option would have a positive impact on criterion b, and a negative impact on criteria a and c. If we do not regard any of the criteria as being more important than any other, we would likely choose UML Activity Diagrams.

Figure 3.4 depicts this notion of method rationale as based on explicating the choices made throughout method construction. The specific example shown is the choice between VIBA Action Diagram versus UML Activity Diagram.

This model of method rationale is explicitly based on the Question, Option, Criteria Model of Design Space Analysis (MacLean et al. 1991). Other approaches to capture method rationale in terms of design decisions are, for example, IBIS/gIBIS[6] (Conklin and Begeman 1988; Conklin and Yakemovic 1991; Nguyen and Swatman 2000; Conklin et al. 2003; Rooksby et al. 2006) and REMAP[7] (Ramesh and Dhar 1992). The process-oriented view of method rationale captured by these approaches is important, especially when acknowledging method engineering as a continuous evolutionary process (Rossi et al. 2004) as will be discussed below in

[5] Material actions are actions that produce material results, such as painting a wall, while communicative actions result in social obligations, such as a promise to paint a wall in the future. The latter thus corresponds to what Searle (1969) termed 'speech act'.

[6] Issue Based Information Systems.

[7] REpresentation and MAintenance of Process knowledge.

Sect. 3.4. However, another, and as we shall see below, complementary approach to method rationale, primarily based on Max Weber's notion of practical rationality, has been put forth as a means to understand why methods prescribe the things they do (Ågerfalk and Åhlgren 1999; Ågerfalk and Wistrand 2003; Wistrand 2009).

According to Weber (1978), rationality can be understood as a combination of means in relation to ends, ends in relation to values and ethical principles in relation to action. Rational social action is always possible to relate to the means (instruments) used to achieve goals, and to values and ethical principles to which an action conforms. Thus, we cannot judge whether or not means and ends are rational without considering the value base upon which we consider the possibilities.

In this view of method rationale, all fragments or components of a method are related to one or more goals (see also Sect. 6.4 on goal-based method construction techniques). If a fragment is proposed as part of a method, it should have at least one reason to be there. We refer to this as the goal rationale of a method. Each goal is, in turn, related to one or more values. If a goal is proposed as the argument for a method fragment, it too should have at least one reason to be included. We refer to this as the value rationale of a method. Figure 3.5 depicts this notion of method rationale, which also includes the idea that goals and values are related to other goals and values in networks of achievements and contradictions. The diagram also includes the actor who subscribes to a particular rationale. Using the terminology introduced above, an actor could be a method creator, a method configurator or a method user.

Each goal is anchored in the method creator's values (Goldkuhl et al. 1998; Ågerfalk 2006; Ågerfalk and Fitzgerald 2006) and goals and values form the essence of the perspective (Goldkuhl et al. 1998) or philosophy (Fitzgerald et al. 2002) of an Information Systems Development Methods (ISDM). Method rationale makes it possible to address the goals that are essential to reaching specific project objectives. Prescribed actions and artefacts, on the other hand, are the means to achieving something (such as the goals). Method rationale can therefore prevent developers from losing sight of that ultimate result and can help them find alternative ways forward. This was clearly evident in Karlsson and Ågerfalk's (2009a) study of method configuration in an agile context and Karlsson's (2013) longitudinal study of the use of method rationale in method configuration.

However, when defining method components in practical SME, Karlsson and Ågerfalk (2009b) suggest restricting the modelling of method rationale to goals only. This suggestion is purely pragmatic and based on the empirical finding that method engineers and developers tend to reason about the purpose of certain method components and often omit discussion of values. It is also important to note that for practical reasons we are not searching for objective goal statements but rather for pragmatic and situated statements that describe the use and effects of method components.

To illustrate how the concepts of method rationale fit together, we will return to the example introduced above. Assume we have a model following Fig. 3.5 populated as follows (assuming that the classes in the model can be represented

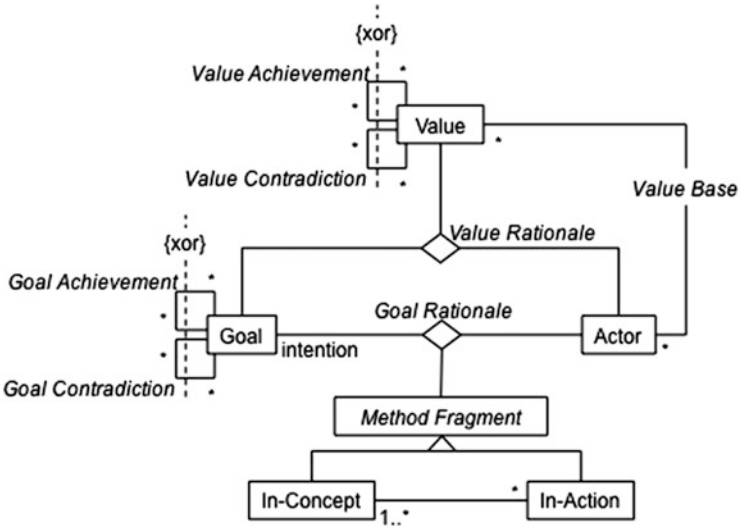

Fig. 3.5 Method rationale as consisting of interrelated goals and values as arguments for method fragments (after Ågerfalk 2006)

as sets and associations as relations between sets, i.e., as sets of pairs with elements from the two related sets). Note that we assume that the actors involved are the creators of the respective fragments, so these are not included in the analysis. This could easily be done and can be used as an additional analytic tool to verify consistency within and across methods with respect to underlying values and how these are reflected in particular method fragments.

A set of method fragments $F = \{f_1$: Representation of the class concept; f_2: Representation of the activity link concept; f_3: Representation of the action result concept$\}$; A set of goals $G = \{g_1$: Classes are represented in the model; g_2: Activity links are represented in the model; g_3: Activity results are represented in the model$\}$; A set of values $V = \{v_1$: Model only information aspects; v_2: Minimalist design of modelling language; v_3: Focus on instrumental v. communicative; v_4: Use well-known formalisms$\}$; Goal rationale $R_G = \{(f_1, g_1), (f_2, g_2), (f_3, g_3)\}$; Value rationale $R_V = \{(g_1, v_2), (g_1, v_3), (g_1, v_4), (g_2, v_1), (g_2, v_2), (g_2, v_4), (g_3, v_3)\}$; Goal achievement $GA = \{(g_3, g_2)\}$; Value contradiction $VC = \{(v_1, v_3)\}$; $VA = GC = \emptyset$.

A perhaps more illustrative graphical representation of the model is shown in Fig. 3.6. If we view each method fragment in the model as possible options to consider, then the goals and values can be compared with the criteria in a structured way. Given that we know that what we want to describe in our notation is a flow of activities (or more precisely the links between activities), we can disregard f_1 outright, since its only goal is not related to what we are trying to achieve. When considering f_2 and f_3, we notice that each is related to a separate goal. However, since there is a goal achievement link from g_3 to g_2, we understand that both f_2 and f_3 would help satisfy the goal of representing visually a link between two activities

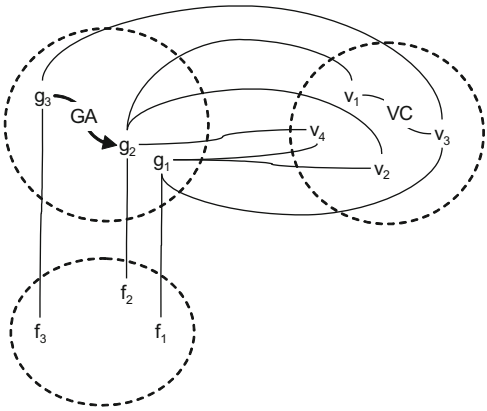

Fig. 3.6 Graphical representation of the method rationale mode showing the tree method fragments, the three goals, the three values and their relationships. The goal achievement relation is represented by an arrow to indicate the direction of the 'goal contribution'. All other relationships are represented by non-directed edges since the direction of reading is arbitrary

(if we model results as output from one activity and input to another, we also model a link between the two), since these two goals are based on different underlying and contradictory values. Since g_2 is related to v_1, and g_3 to v_3, we must choose the goal that best matches our own value base. This could and should be expressed by the criteria we use. If, for example, we believe that it is important to direct attention to instrumental versus communicative aspects (v_3), then we should choose g_3 and consequently f_3. If, on the other hand, we are only concerned with modelling information flows, then g_2 and consequently f_2 would be the option to choose.

Empirical observations show that the method component's overall goals and artefacts are important during method configuration (Karlsson and Ågerfalk 2004) and hence they are part of the interface. An artefact, as discussed above, is designated as an input and/or a deliverable (output). This is necessary in order to deal with the three fundamental actions that can be performed on an artefact: create, update or delete. In cases where an artefact is only created by a method component it is classified as a deliverable. If the artefact can be updated by the same method component, it is also classified as an input. Furthermore, a component can have one or more input artefacts, but has only one deliverable (which thus defines the 'layer of granularity' of the component).

The concept of method rationale described above applies to both construction of methods and refinement of methods-in-action (Rossi et al. 2004). Since method descriptions are means of communicating knowledge between method creators and method users, it could be used as a bridge between the two and thus as an important tool in achieving rationality resonance, as discussed above.

From the earlier example in this section, we can see that method rationale is related to both the choices we make during method construction and to the goals and values that underpin the method constructs we choose among. The example used above was at a very detailed level, focussing on rationale in relation to method fragments at the concept layer of granularity. The same kind of analysis could be performed at any layer of granularity and may consider both process and product fragments (i.e., both activities and deliverables). As an example, let us consider the

use of agile methods for globally distributed software development. This may seem counter-intuitive in many ways. One example is that agile methods usually assume that the development team is co-located with an on-site customer present at all time (Beck 2000). By analysing the reasons behind this method prescription (i.e., the suggestion by the method creator), we may find that we can operationalise the intended goals of co-location (such as increased informal communication) into other method prescriptions, say utilising more advanced communication technologies. In this way, we could make sure that the method rationale of this particular aspect of an agile method is transferred into the rationale of a method tailored for globally distributed development. Thus, we may be able to adhere to agile values even if the final method does look quite different from the original method. That is to say, the principles espoused by the method creators may be logically achieved to the extent that they are relevant in the particular context of the final situational method.

It is important to see that method rationale is present at all three levels of method abstraction (Fig. 3.3): ideal typical, situational and in-action. At the ideal typical level, method rationale can be used to express the method creator's intentions, goals, values and choices. This serves as a basis for method configurators (i.e., those who tailor a situational method) and developers in understanding the method and how to tailor it to best advantage. In the communication between configurator and developer, method rationale would also express why certain adaptations were made when configuring the situational method. If we understand different developers' personal rationale, we might be able to better configure or assemble situational methods.

Combining the two aspects of method rationale gives us a structured approach to using method rationale both as a tool to express and document a method's rationale, and as a tool to analyse method rationale as basis for method construction, assembly, configuration and use.

3.4 Evolutionary Method Engineering

Method engineering involves a learning process in which the current level of expertise and the situation influence the outcomes (Hughes and Reviron 1996). Thus, any organisation that develops systems not only delivers them but also learns how to perform system development and to mobilise associated knowledge (methods), but also improves their development capabilities (learning by doing). The development organisation builds its knowledge about how its methods work in certain development situations. These experiences complement the codified method knowledge and should lead into better applications of the method in the future. Checkland (1981) was an early advocate of a learning-based approach to method development through cyclical action research. In this view, evolutionary method engineering is seen as a continuous refining process. However, as Lyytinen and Robey point out (1999) large parts of this experience are lost since the experiences

are seldom collected and interpreted—thus emphasising the importance of retrospectives (Kerth 2001).

Methods can never gather all previous knowledge and anticipate all future development situations. Therefore, it is fruitful to view methods from an organisational learning perspective. This perspective analyses system development situations and the role of methods through 'reflection-in-action' (Schön 1983). In Schön's view, a large part of a designer's knowledge of ISD is a result of his or her reflections of the situation, rather than being determined by the methods. In real life, the method-in-action is adapted and interpreted by designers based on their understanding of the events and contingencies. At the same time the current version of the method is a result of those reflections so that designers' tacit understandings are made explicit so that they can be made understandable to others (Nonaka 1994).

Rossi et al. (2004) claimed that reflection-in-action and technical-rationality are complementary in systems development and both explicit and tacit knowledge are needed to develop systems. Thus, a good method should adapt to a situation and provide cognitive frames and norms that designers can use, but also challenges the use of their experiential knowledge (Argyris and Schön 1978). Such a learning view has been called evolutionary method engineering (Tolvanen 1998). Because evolutionary SME aims to continually improve ISD methods it can be regarded as a learning process in which individuals (Schön 1983), communities and organisations (Nonaka 1994) create, memorise and share knowledge about system development through codifying it to methods. This double loop learning leads to continuous modification and augmentation of an organisation's methods. In evolutionary ME, method evolution is seen to be necessary since organisations have to deal with different method versions for different implementation targets and development contexts (as for example with UML (OMG 2010)).

Two different types of method evolution have been identified (Rossi et al. 2004): changes to methods reflecting general requirements of changed technical and business needs, and those relevant to the ISD situation at hand. The former relates to the general genealogy of methodical knowledge within the method developer and user community, and the latter with how these general evolutions are adapted into local situations and affect development practices. We can anticipate that user-centred method engineering calls for extensive local modifications and possibilities for evolutionary variants of methods.

3.5 Method-User-Centred Method Engineering[8]

A problem in situational method engineering, similar to software and systems engineering, is that requirements, here method requirements, need to be specified and managed. Evolutionary method engineering addresses the management problem by allowing method requirements to evolve over time. The initial specification

[8] We acknowledge contributions of Dr. Fredrik Karlsson to this section.

of method requirements calls for specific techniques. One such technique, as partly explored in relation to the MC Sandbox tool, has been borrowed from user-centred design and termed method-user-centred method engineering (Karlsson and Ågerfalk 2012). These authors provide extensive discussion about method-user-centred method engineering along with a case study on the use of these ideas as implemented in MC Sandbox (see also Sect. 7.3.2).

3.5.1 Method Requirements

Methods exist for the purpose of supporting project members during development projects. These people are users of the method in the same sense that end-users are users of software. Hence, method users impose requirements on methods in much the same way that end-users have requirements on information systems. The actual content of requirements engineering processes varies, although often the core activities include elicitation of problems and solutions, negotiation of problems to solve and solutions to adopt as well as commitment to implement the selected solutions. The requirements are developed during these activities and a number of challenges are evident.

Firstly, method requirements are not always clear, neither to the method engineer nor to the method user—partly because the systems development task is not always well understood, partly because the project members' method varies. The first problem indicates dependencies between systems development and method engineering, which is why we discuss situational methods in the first place. These dependencies are not always possible to identify completely initially. Rather they become visible incrementally, which is also acknowledged in incremental method engineering as discussed in Chap. 7. The second problem illustrates the necessity to improve the communication about what is possible to achieve and reasonable to expect from the method at hand. Method users will learn about the possibilities offered by the method and discover new requirements as a project progresses.

Secondly, there is not just one set of requirements, since requirements are by nature emergent and constantly negotiated and renegotiated (Chakraborty et al. 2010; Holmström and Sawyer 2011). Different stakeholders with different interests typically bring their own set of requirements to the table. Klein and Hirschheim (2001) emphasise the importance that these different interests are 'understood and debated'. Depending on the selection of method users, the method requirements are therefore likely to be different. Developing a shared understanding is therefore of prime importance (see earlier discussion of Fig. 3.2).

It is also true that not all stakeholders have the same power and possibility to influence the requirements process (Coughlan et al. 2003). In any case, these different sets of requirements can of course be (at least partly) conflicting. In some cases, apparent conflicts are based on misunderstandings that can be solved through clarification. Conflicts can also arise due to differences in perspectives and what is perceived as important during the project. Stakeholders may not share the same value base, as discussed in Sect. 3.3 above.

Finally, project resources may not allow for all requirements to be considered. A situational method will thus only solve stakeholders' needs and problems to a certain degree. Thus, the method requirements process must include ways to handle method requirements conflicts and requirements viewed as negotiated commitments to be fulfilled during the project.

3.5.2 Why Method-User-Centred?

In software and systems engineering, end-users have to conceptualise, explicate and negotiate their requirements; creativity has to be stimulated in that process (Maiden et al. 2004). Malcolm (2001) suggests that user-centred approaches are especially appropriate when addressing tacit, semi-tacit and future systems knowledge. Arguably, mental models and tacit knowledge are as crucial to successful method engineering as to systems design. Stolterman (1992) addressed the importance of understanding the method creators' mental model of their created method. Stolterman and Russo (1997) use the terms public and private rationality for this purpose. Public rationality is the intersubjective understanding of prescribed actions and results, and about why a specific part of a method is prescribed. This is therefore what we refer to as method rationale in this book. Private rationality is expressed 'in the skills and in the professional ethical and aesthetic judgments' of a person (Stolterman and Russo 1997). The method creator has to influence not only public rationality but also the private rationality of the method user. Otherwise, method users may not be able to use the method to its fullest potential. Consequently, it is important to involve method users early when crafting a situational method. Just as when involving end-users early in systems development, this involvement should focus on method-user-centred aspects.

3.5.3 Bringing User-Centred Ideas to Situational Method Engineering

Gould and Lewis (1985) proposed three principles that are included in what we today call user-centred design: (1) early focus on users and tasks, (2) empirical measurement and (3) iterative design. According to Cato (2001), it is possible to conceptually view user-centred design as a triad: the user, the use and the information. This triad focusses on who is using the technology, how technology is used and what is required to support that use. Translated into situational method engineering, we should thus focus on who the method users are as a team and these users' needs during a project (i.e., what kinds of challenges are found in the project), and how methods are used in the organisation. Furthermore, designing a situational method is an iterative process where the method is continuously evaluated and, if necessary, changed. Although a user-centred approach shifts the emphasis in software development from technology to people, Constantine and Lockwood (1999) stress that it should be more than this—that we should focus on

usage rather than user. In their usage-centred design (UCD) approach (Constantine 1996), they advocate five key elements:

- Pragmatic design guidelines
- Model-driven design process
- Organised development activities
- Iterative improvement
- Measures of quality.

Storyboarding (Higgins 1995) and prototyping (Boar 1984) are techniques frequently used in user-centred approaches to create a feel for a proposed solution (e.g., Carroll 1994; Hall 2001) and to visualise commitments made. The idea is to make the design more tangible by letting use-scenarios and visualisations drive the design process. Visualisation often starts with low-fi prototypes, which make it possible to identify potential problems early and at a low cost (Rettig 1994). A paper-based storyboard typically captures the structure, possible navigation through the information system, information provided by the system and by the user and the result of users' actions (Cato 2001).

Nickols (1993) emphasises that a prototype is a working model that is subject to negotiation. It therefore does not have to be complete in terms of functionality. Low-fi and high-fi prototypes differ in the sophistication of their technical implementation and the cost of change. Low-fi prototypes implement less technical complexity and are hence less expensive to change. They also implement less functionality. High-fi prototypes, on the other hand, typically implement more complex functionality and are therefore more accurate but also more costly to change.

Transferring these basic ideas to situational method engineering shows that visualising the method design and its parts is essential to SME. Prototyping also involves a continuous evolution of the prototype and its design. Naumann and Jenkins (1982) present prototyping as consisting of four activities: identify basic requirements, develop a working prototype, implement and use and revise and enhance. The two latter activities are performed iteratively, somewhat similar to evolutionary method engineering (Sect. 3.4) and scenario-based approaches (Rolland et al. 1999). It is not surprising, then, that the implementation of evolutionary method engineering in a tool like MetaEdit+ (Sect. 7.3.1) shares several characteristics with high-fi prototyping. Consequently, evolutionary method engineering could be complemented with an approach where the method users are involved more directly in the initial tailoring of the method. However, this may be difficult to achieve since method engineering tends to be a detailed process, especially if high-fi prototypes, such as runtime CASE-tool implementations of methods, are brought into the equation.

As a complement, it may therefore be fruitful to use also low-fi prototypes and storyboarding in situational method engineering as suggested by Karlsson and Ågerfalk (2012) and implemented using MC Sandbox as described in Sect. 7.3.2. This approach combines the idea of visualising the situational method as a storyboard by reducing the amount of detail. Clearly, that approach shares similarities with the map construction approach presented by Rolland et al. (1999)—see

Chap. 4. The focus of method-user-centred method configuration is on what method parts add value to the development project and its members as a team. Hence, it moves away from the use of complex meta languages when working closely with method users, similarly to the way prototypes are used in discussions with end-users instead of complex diagrams and source codes. The underlying idea is facilitate method users to formulate their requirements, debate them as a team and explicate their commitments. Essentially, the use of prototyping and storyboarding facilitates the negotiation of several mental models and makes implications tangible. The prototyping tool can at the same time act as a documentation tool during elicitation and negotiation of method requirements.

Altogether, these ideas affect the concepts, the models and the meta methods that we use in situational method engineering. Even more importantly, they affect the tools that we use in the process. Essentially, tools have to support the simplification of method modules to emulate a low-fi prototype. Still, they must provide the information needed for discussing and negotiating method support and potential results of different choices. MC Sandbox, as described in Sect. 7.3, is explicitly designed to deal with these constraints but certainly other tools can be used to achieve similar effects.

3.6 Summary

This chapter has highlighted that situational method engineering is a social process that needs to pay attention to human factors such as values, attitudes and knowledge. Method rationale has been presented as a way to understand how methods encapsulate rationality and how different stakeholders may perceive a method differently. Evolutionary method engineering and method-user-centred method engineering were introduced as two current approaches that aim to take human and social aspects of SME into account by acknowledging that method requirements are constantly renegotiated and evolving. The two approaches have been shown to be complementary and could be used together in order to properly address the social aspects of SME in practice.

http://www.springer.com/978-3-642-41466-4

Situational Method Engineering

Henderson-Sellers, B.; Ralyté, J.; Ågerfalk, P.; Rossi, M.

2014, XX, 310 p. 167 illus., 58 illus. in color., Hardcover
ISBN: 978-3-642-41466-4