Handbook of Software Engineering and Knowledge Engineering
Vol. 2

SUPPORTING SOFTWARE PROCESSES USING KNOWLEDGE MANAGEMENT

RALF KNEUPER
Transport-, Informatik- und Logistik-Consulting (TLC) GmbH
Kleyerstrasse 27, 60326 Frankfurt/M, Germany
Email: ralf.kneuper@gmx.de
URL: http://www.kneuper.de

In this paper we describe how software processes can be supported using knowledge management, concentrating on the creative manual tasks performed by developers. The main support needed by developers is the availability of the knowledge about the development processes relevant to their task. As a result, software process modeling can be considered as a knowledge management task.

The paper provides an overview of the knowledge needed by developers. It then reviews various approaches to knowledge representation including artificial intelligence, structured plain text, and software process modeling approaches, to identify representation schemes suitable for software processes. Finally, the paper looks at the management aspects of software process knowledge.

Keywords: Software process; Software process modeling; Knowledge management; Knowledge representation;
XML

1. Introduction

1.1. *Software Processes*

Software processes are all the processes that are used to create software, including analysis, programming, configuration management, etc. Based on [49], we distinguish various kinds of software processes, see Fig. 1.

Software processes can be explicitly defined, or implicit processes that are used without thinking much about the process as such.

Reasons for defining (modeling) software processes are [24, p. 5][53]:

- defined processes put structure into software development
- good software processes support developers in their work. Using the software process, developers know which step to take next, and they have templates for the results to be produced. As a result, they can concentrate their creativity on solving technical problems, rather than re-inventing the wheel, and solving problems that are caused by unclear cooperation with their colleagues.
- they form a basis for improvement and learning, including process monitoring and process simulation as well as process validation and verification.

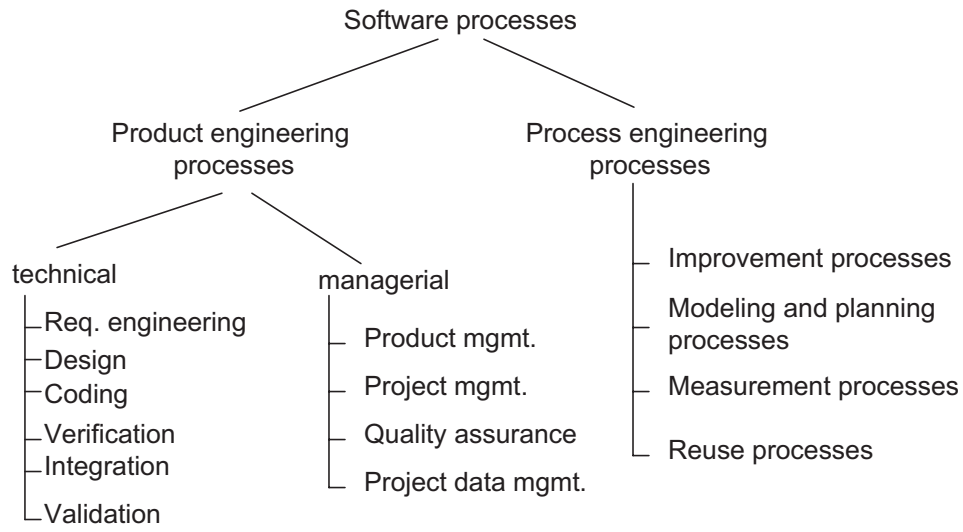

Fig. 1. Software processes (based on [49])

- they make development more predictable. This is particularly important from the management perspective, since it helps to estimate correctly the time and cost needed for developing a system, and the resulting product quality.
- computer-supported software processes additionally allow process enforcement, automation, and guidance.

Examples of defined software process models are the (Rational) Unified Process (RUP, see [38]) or the German V-Model (see [56, 52]), which both define fairly complete software process models including the various phases as well as supporting activities such as project management and quality assurance. On a rather coarse-grained level, there are various process models (also called phase models or life-cycle models) such as the waterfall model [51] or the spiral model [13] (see [15] for an overview of the different life-cycle models).

These models define the top level of a process model hierarchy (see Fig. 2). An organization typically does not use such a standard process model "as-is", but tailors it to its own special needs, assigning roles to departments, adding information about tools to be used for some process steps, adding specific or leaving out irrelevant steps. From this organizational process model an individual project then derives its project plan, again leaving out irrelevant steps (e.g. all steps that concern the creation of a database if the application does not include a database) and adding (project-) specific steps (e.g. create some artifact only required by this customer), and also iterating steps as appropriate (e.g. repeating the step "describe dialogue" four times since there are four dialogues in the application). Finally, of course, a project plan must include information about effort planned for the individual tasks,

milestones, etc.

This does not imply that all three levels must always be present. Many organizations define their process model directly, without recourse to any standard model. Occasionally, a project plan is derived directly from some standard model, for example because the customer demands the use of this model. Many project plans are not based on any process model at all — this is quite common but falls outside the scope of this paper. And (last and least) some projects are run without any project plan at all.

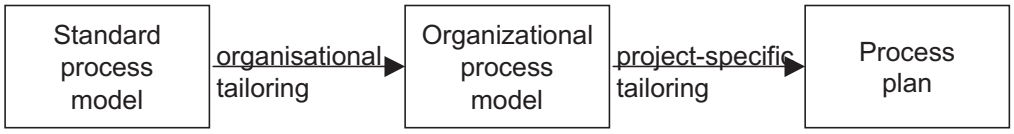

Fig. 2. Process model hierarchy

Current issues in software processes research are (cf. [24, Sec. 1.8])

process formalisms and process modeling languages focused on creating notations to describe software processes, usually to be "read" by various tools used for

- analysis, such as consistency checking ("Is every artifact that is used by an activity created by an earlier activity?" "Is every defined role assigned at least one task?") or for improvement "Which of these two ways of achieving a certain result is more efficient?")
- process support, such as tailoring a general process to a specific project, or process workflow support

process-centred environments, called *process-sensitive software engineering environment* (PSEE) by [24], go one step further, integrating the various development tools into one environment under the control of some form of process engine. E.g. if a developer is assigned the task to create an entity/relationship model and starts this task, the appropriate editor for entity/relationship models is started up automatically.

software process improvement (SPI) is concerned with ways to define concrete process models for an organization, to introduce them into the day-to-day work of the organization (which tends to be much more difficult than creating the model in the first place), and finally to continuously improve and adapt these models.

In contrast to the previous two items, SPI has little technical contents but focuses on soft factors such as management, organizational change, psychology, etc.

Probably the best known approach to SPI is the Capability Maturity Model (CMM) [18, 52, 53] (`http://www.sei.cmu.edu/cmm/cmm.html`), which defines five maturity levels and thus provides an improvement path for organizations developing software. SPICE [50] is based on similar ideas and working on an ISO standard 15504 [35] for assessing the maturity of software developing organizations.

An approach specifically targeted at introducing CMM is IDEAL (Initiating, Diagnosing, Establishing, Acting, Leveraging) [41].

Other approaches to SPI include Basili's Experience Factory and the Quality Improvement Paradigm [7, 8, 9] as well as approaches from quality management in general, not specific to software, such as ISO 9000 (with its software-specific interpretation, ISO 9000-3), Kaizen, and Total Quality Management.

definition of software process, finally, is concerned with creating concrete software process models, either as a general standard, such as the German V-Modell, as a de-facto standard such as the (Rational) Unified Process, or as an internal, company-specific process model (see e.g. [6, 43]). ISO 12207 [34] provides a lot of input on the contents of such a process model.

A good overview of the topic of software processes and the research questions involved, covering mainly process formalisms, process modeling languages, and process-centred environments, is given by [24] and [49], the current state of the art can be found in the proceedings of the European Workshop on Software Process Technology series (EWSPT, e.g. [27, 21]) and the International Conference on the Software Process series (ICSP).

More information on SPI can e.g. be found in [49], or, with a strong CMM bias, on the SEI web pages (`http://www.sei.cmu.edu`), or the proceedings of the annual SEPG (Software Engineering Process Group) and European SEPG conferences.

Another way to view software processes is that they are a form of knowledge. Managing software processes is a special case of knowledge management. The current paper follows this view to identify what can be learned from knowledge management for the support of software processes.

The remainder of this introduction will analyze how much support software process technology can give (see Sec. 1.2), and then define the basic concepts of *knowledge* (see Sec. 1.3) and *knowledge management* (see Sec. 1.4).

After that, in Sec. 2, we discuss the knowledge that is needed in the special case of software processes. Sec. 3 describes some approaches to knowledge representation and looks at their suitability for the kind of knowledge considered here. Since dealing with software process knowledge is a management task even more than a technical problem, Sec. 4 looks at the management aspects of process knowledge and their implications on software process technology. Finally, in Sec. 5, we draw some conclusions.

1.2. *Technology to support software processes*

Software development consists of a number of different kinds of tasks including

- creative tasks,
- administrative tasks, and
- communication tasks.

A common feature of these tasks is that they require a fairly large amount of knowledge.

Software process technology should therefore emphasize support for human developers in their task.

The obvious question now is what software process support human developers need for software development. The suggestion of this paper is that the main support needed is to provide *knowledge* to the developers, e.g. in the form of descriptions of tasks and artifacts (results) (cf. Sec. 2.2), or, more advanced, in the form of advice and guidance. Phrased differently, software process support should be considered as a *knowledge management* task [23].

Knowledge *management* is quite a different concept from knowledge *representation* as a topic of Artificial Intelligence, even though there is some overlap. Knowledge management is a management task mainly dealing with humans, while knowledge representation is concerned with formal representation of knowledge typically used for certain algorithms (such as reasoning and planning).

This is very different from the current paradigm in software process technology which concentrates on the attempt to automate software processes [49], as suggested in particular by Osterweil [46]. The basic assumption behind this approach is, obviously, that software development contains large tasks that are sufficiently simple to be formalized, or, put differently, it restricts attention to such tasks (compare [57]). This emphasis on automation and formalization results in a number of requirements that are important for the industrial use of software process support but are not satisfied by many software process support tools (see [36]).

The difference between the two paradigms (software process support as automation of software processes or as support for human knowledge workers) is not just of academic or scientific interest but has practical implications since software process models, like all models, are a form of reality construction. They are not neutral descriptions but have a purpose or bias. As a result, they help to create a mental model of software processes, be it as creative, mental processes or as a mechanizable, menial task. Such a mental model (or paradigm, see [39, Postscript Chap. 4]) influences the way we see the world and interpret the external stimuli. This is one of the reasons why it is important to have a paradigm that leads in a useful direction. Conradi et. al. in [20] argue similarly that automation in this context is of limited value. About software process research, they say:

> "Quite often, we want to provide automatic support to activities that simply do not need to be supported 'that far'. For instance, while it is

> certainly useful to clearly describe the steps in a design method, it is purposeless to formally model it to support its detailed enactment (e.g., by indicating which steps in the method have to be carried out and in which order). Developers will never accept a tool that operates at that level of detail. Moreover, it would be useless. You can use a design method effectively only if you 'know' it so deeply that it is part of your mental habit, and not just because 'someone else' mandates what you are supposed to do at any time." [20, pp. 102f]

Quality management approaches such as CMM and ISO 9000 also move towards putting more structure into the software development process. However, this is not to be confused with automation of the processes — these approaches are mainly concerned with defining the manual processes, describing the steps to be performed (such as reviews) and the responsibility for these steps.

Coming back to the question of support of software processes using knowledge management: necessary tasks in the use of software processes that could be supported by knowledge management are

- define processes (knowledge elicitation and representation)
- introduce processes into the day-to-day work of the developers — just defining the processes and making the definition available is rarely enough
- apply or live a process
- improve and adapt the processes since they will never be perfect, and the environment keeps changing (new technology, better qualified developers, etc.)
- tailor and instantiate the processes to turn them into a project plan

Since the tasks in this list are processes that work on the software processes, they are also called meta-processes. These meta-processes need to be closely coordinated with the software processes themselves since the meta-process can lead to a change of the process, possibly while the process is running (e.g. a process improvement that is introduced during the lifetime of a project, see [48]).

1.3. *What is Knowledge?*

Following Davenport and Prusak, we distinguish the concepts of data, information and knowledge. The following definitions are based on [23, pp. 2–6]:

Data are a set of discrete, objective facts. A typical example of data in software process modeling is "an entity-relationship diagram consists of boxes representing entities and lines between the boxes, representing relationships between entities".

Information is data that 'makes a difference' to the receiver. Put differently, information is data with added value, e.g. by putting it into context, by mathematical or statistical analysis, or by summarizing it in a more concise form. In

software process modeling, examples for information are "developing a software system consists of the following tasks .... In task X_1, the artifact Y is produced and used as input for task X_2".

Knowledge "is a fluid mix of framed experience, values, contextual information, and expert insight that provides a framework for evaluating and incorporating new experiences and information" [23, p. 5]. Knowledge can be considered as information with added value, such as the relationship between different pieces of information, or the implications of that information on decisions and actions. Examples of knowledge in software process modeling are "Step A is not needed for Java applications developed for customer C." "If you have experienced developers, it is better to do step A before step B, while inexperienced developers have to do it the other way round." "Artifact Y is very important for complex applications but superfluous for simple applications." Part of the difficulty of applying such knowledge is that it depends on judgements about the current situation (Is developer D experienced? The answer to this question depends both on the developer and the tasks at hand — a developer may be an experienced Cobol programmer but know nothing about Java.)

Other authors, especially in the AI community, e.g. [26], define the term "knowledge" much wider, including what we called information. This leads to different answers to the question whether in software process modeling, we are dealing with *knowledge* or whether it really is *information* (or even data). According to the definition used here, a software process model itself does not contain knowledge but (at best) information. The challenge now is to present the model in such a form that the users (developers) are able to turn this information into knowledge — they have to "understand" the model. This implies providing sufficient information about context and implications, but also representing the model in a way such that it is easy to understand.

The same statement actually applies to any form of knowledge representation, whether in the form of a knowledge base, text book, process model, or other. The very act of representing knowledge turns it into data or information. Providing more such information about context and implications does not necessarily help to make this transition to knowledge; on the contrary, too much information can lead to information overload and the information turns into data since users can no longer cope with it.

1.4. *Knowledge Management*

Knowledge management is the management discipline that is concerned with creating, preserving and applying the knowledge that is available within an organization. Typically, this is done using techniques such as training, process modeling, experience (data) bases, and networking of experts. For a very practical, hands-on report on applying knowledge management at a software house see

e.g. [17]. For a large collection of references on knowledge management see e.g. `http://www.bcs-sges.org/kmreport/bibliography.htm`.

As stated above, knowledge *management* is quite a different concept from knowledge *representation* as a topic of Artificial Intelligence, even though there is some overlap. Knowledge management is a management task mainly dealing with humans, while knowledge representation is concerned with formal representation of knowledge typically used for certain algorithms (such as reasoning and planning).

Hansen, Nohria and Tierney [28] distinguish two different approaches to knowledge management based on the importance they put on formalization:

the codification strategy puts the main emphasis on making tacit knowledge of the organization explicit, e.g. in a process model.

the personalization strategy concentrates on support for person-to-person knowledge transfer, leaving knowledge tacit but making it available by communication between people. A typical technique here is to describe *who* knows about a certain topic, rather than describing the topic itself in sufficient detail for someone else to read and apply it.

The main focus of the current article is on software process modeling and therefore on the codification strategy. Depending on context and the task at hand, either strategy can be useful.

However, it is important to keep in mind that knowledge essentially is human-based. If people (such as developers) need knowledge to solve a problem, they usually tend to ask their colleagues rather than search a database. The challenge for software process technology is to make the tools sufficiently useful for developers to prefer to use the tools rather than asking their colleagues.

For common tasks, this is a (within limits) achievable goal because it is more efficient to describe a common task once and use the description many times afterwards. For uncommon or very difficult tasks, this is not true and it makes more sense to direct the developer to an expert in the field under consideration. Therefore, software process models should not only provide information about the processes themselves but also about the experts to ask when going beyond the limits of the defined process.

Probst [47] has defined a structure for knowledge management, called the "building blocks" of knowledge management, see Fig. 3.

In the special case of software processes, these building blocks essentially repeat, in a more structured way, the process support tasks mentioned in Sec. 1.2 above. We will look at them in more detail in Sec. 4.

2. Knowledge Needed

In this section, we describe the knowledge needed to support software processes in more detail.

Before we can do so, we need to define *who* needs the knowledge we are talking about. First of all, this includes the developers themselves and their project leaders,

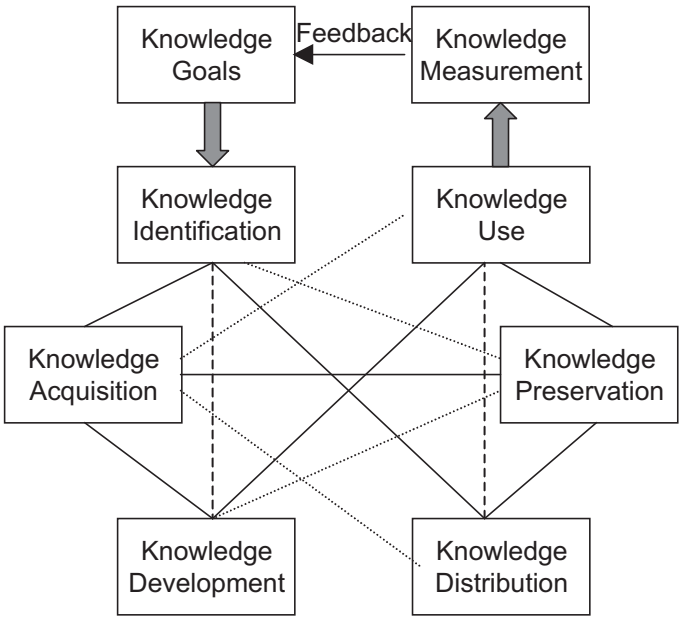

Fig. 3. Building blocks of knowledge management [47]

but also supporting personnel such as technical writers who write the user documentation, quality assurance personnel, etc. Furthermore, for customer-specific software (as opposed to off-the-shelf software) the customer of the software developed will often require knowledge about the processes used before signing a contract, and during a project to be able to provide or sign-off input such as requirements or domain knowledge. Similarly, management needs knowledge about the processes used in order to judge and optimize their effectiveness.

The knowledge needed includes an overview of the processes used (Sec. 2.1, descriptions of the tasks to be performed and the artifacts to be produced (Sec. 2.2), the development method used (Sec. 2.3), knowledge about best practices and lessons learned (Sec. 2.4), and other knowledge needed for software development (Sec. 2.5).

2.1. *Overview of Processes Used*

First of all, everyone involved needs an overview of the software processes used. This describes the type of life cycle used (waterfall, V-shaped, incremental, spiral, ...) as well as the phases in the life cycle together with their main goals, activities and results.

While for some groups, such as the customer, this may be all the knowledge needed about the software process, most users will need the overview to identify where their own piece of work fits in, which interfaces to other people or groups they have in the development process, to set priorities in case of conflict, etc.

2.2. *Description of Tasks and Artifacts*

This is the core part of software process support and includes templates for the various artifacts (such as a project plan or an entity-relationship model) to be developed as well as descriptions of the steps to be performed for a task, plus their pre- and post-conditions, and checklists to ensure completeness of the tasks.

Pre- and Post-Conditions Pre-conditions of a task are conditions that have to be satisfied at the beginning of the task, typically the previous existence of a certain artifact or the previous performance of a certain other task. For example, before programming of a component is started, the specification of the component should be complete.

Post-conditions, on the other hand, are conditions that are guaranteed to be satisfied at the end of a task, such as the existence of a certain artifact created by the task. Another typical usage of pre- and post-conditions concerns the status of an artifact. E.g. the pre-condition of a review typically requires the existence of the result to be reviewed in status "draft" or similar. The post-condition states that the contents of the artifact has not changed, but that the artifact now has the status "accepted" or "rejected".

These dependencies between different tasks are one of the main difficulties in tailoring a process model to a specific context, since if one adds or deletes a task, all pre- and post-conditions of other tasks still have to be satisfied.

Knowledge about pre- and post-conditions is used in planning a project (pre- and post-conditions impose a partial order on the tasks to be performed, or even imply that a certain task needs to be added in order to satisfy the pre-condition of a later task) as well as in performing a given task, since the post-condition succinctly describes what is to be achieved by the task ("create artifact X"; "promote artifact Y from status 'draft' to status 'released')".

Finding a suitable sequence of tasks such that the pre-conditions of all tasks are guaranteed to be satisfied by post-conditions of earlier tasks is a typical example of a planning problem in AI [26, 30].

Description of Steps and Artifacts The description of the steps to be performed and the artifacts produced is the central part of software process support for developers. This description typically includes the methods(s) used, any sub-steps, and templates for the artifacts produced.

Usage of these descriptions of steps and artifacts can vary a lot between developers: let's say they have a certain task, such as the specification of a dialogue with given functionality. Then depending on their experience, developers may follow the process described step by step, or they just get the template for the artifact needed and fill in the blanks. One of the problems in the latter case is, of course, to ensure that the developer at least gets the current version of the template and does not reuse an older one. A necessary, but not sufficient,

condition for this is to make it very easy to get the template from the standard repository.

Other developers may just want to review the main steps to remind themselves of the general process, e.g. because they have not performed this particular task for some time, while yet others just need the details of one particular step. Software process technology has to support all these different uses and therefore present the relevant knowledge on different levels of detail.

2.3. *Development Method Used*

For the development method used, essentially the same applies as for the tasks and artifacts described above, since a development method can be described as a collection of tasks and artifacts. The main difference is the different level of abstraction: a method tends to be more high-level (when to perform a certain step?), while the description of a single task or artifact tends to contain a lot more detail.

2.4. *Best Practices and Lessons Learned*

Best practices and lessons learned involve knowledge about approaches, techniques, etc. that have proven useful in other contexts and might therefore be helpful for the developer to know about, even though it has not (yet) been expressed as a general process. Best practices can come from within the own organization or from the outside, i.e. practices that are used in other companies.

Knowledge about best practices and lessons learned is typically presented in the form of case studies, since it has not been or cannot be made fully explicit or sufficiently general to be included in the defined software processes. See [44, p. 55f] or [45] for some examples of knowledge bases containing lessons learned and best practices.

[45] also describes a number of difficulties with best-practice knowledge bases:

- Where do you find knowledge about best practices? This is often not publicly available but has to be searched out.
- Best practices are constantly changing. Even if you have well-documented, useful best practices today, the same practices might be completely inadequate tomorrow, due to technology changes or market changes (e.g. time-to-market or quality level of the product is no longer acceptable).
- An active approach is needed; it is not sufficient to just gather knowledge about best practices, but best practices have to be created in the first place.

All these difficulties apply to all process model knowledge bases, not just to those containing best practices.

2.5. *General Software Engineering Knowledge*

The following knowledge is also needed for software development but not usually considered part of software process models. The main difference between the knowledge listed here and the knowledge contained in a software process model is that the process model describes or prescribes the steps performed and the results created. The knowledge listed here helps the developers but does not state what they have to do.

[42, p. 292] list the following relevant software engineering knowledge:

- domain knowledge
- requirements knowledge
- design knowledge
- implementation knowledge
- programming knowledge
- quality factors
- design rationale (see the Chapter on Rationale Management by Dutoit and Paech in this Handbook)
- historical knowledge

Other knowledge needed includes

- development and target environment, tools: this is an aspect that is quite important for developers, although it is not really part of the software process as such. During their development tasks, developers need reference material on issues such as the programming language and the compiler, the DBMS used, middleware and operating system (for development and target environment), etc. This reference material has to be integrated into the development environment so that for any development process step, the necessary reference information is immediately available.
- measurement data: in order to continually improve the software processes, measurement data about properties such as software size and cost, productivity, number of defects found, etc., must be stored and made available.
- developer skills
- support information and typical problems
- market data (mainly, but not only, needed for creating off-the-shelf software products, as opposed to customer-specific software).

This kind of knowledge has to be handled by knowledge management [17] but will not be considered in this context since it is not really part of the software process itself.

2.6. *Administrative information*

In addition to the knowledge listed above, some administrative information needs to be stored with the process knowledge, such as author or owner of a process, release date, version, scope or area of applicability, etc. In this context, we will ignore such administrative information.

3. Knowledge Representation for Software Process Support

In the following we will analyze some approaches to knowledge representation and their usability for the kind of knowledge described above. Based on an analysis of the requirements on such a knowledge representation notation in Sec. 3.1, we will look at AI approaches to knowledge representation (Sec. 3.2), software process modeling notations (Sec. 3.3), business process modeling notations (Sec. 3.4), and various forms of structured plain text (Sec. 3.5). In the summary (Sec. 3.6), a short comparison between the different approaches is given.

One should note that strictly speaking, we are not really talking about *knowledge* representation since, as described above, only people can have knowledge, not machines. If knowledge is represented in some form of knowledge repository, it turns into data or, at best, information. To turn these data back into knowledge requires a human who is able to interpret the data and make sense of it. However, following standard usage we will still talk about knowledge representation.

Many knowledge representation notations were created for use by automated reasoning and inference engines and not really designed to support humans in their work. Not surprisingly, they often turn out to be unsuitable for this task, partly because they do not support free text, explaining e.g. the steps to be performed to create a certain results.

3.1. *Requirements on Knowledge Representation*

Finlay and Dix [26, Ch. 1.4] suggest the following characteristics to assess knowledge representation schemes. Note that these criteria were originally intended for judging knowledge representation notations in artificial intelligence, to be used for automated deduction and similar algorithms, and are therefore less well-suited for judging notations for knowledge representation for human users. The definitions have therefore been slightly adapted for our purposes.

expressiveness: a representation notation must be able to represent the various types of knowledge we wish to represent, on the appropriate levels of granularity.

The types of knowledge we want to represent in order to support software processes have been described in the previous section. One of the difficulties lies

in the necessary combination of well-structured knowledge, such as "activity A takes artifact X_1 as input and produces X_2 as output", with ill-structured knowledge such as "to create a class model, it is usually best to start by ...". As a result, a notation that is used to represent knowledge for human users must allow ill-structured descriptions, i.e. free text.

effectiveness: for structured knowledge, effectiveness implies that a representation scheme must provide a means for inferring new knowledge from old. Since in our case the emphasis is on making knowledge available to the human user rather than an inference engine, effectiveness means that users can find and use the knowledge needed for their work. This implies that appropriate search and selection mechanisms must be available, see below.

One way to make a knowledge representation notation more effective is the possibility to provide different views on the knowledge represented, e.g. for different groups of users (developer, manager, customer, automated tool, etc.) or for different kinds of projects (structured or object-oriented development, large or small project, high-risk or low-risk, etc.).

efficiency: in our case, efficiency means that the user must be able to find the knowledge needed within a reasonably short time frame, and be able to find and use it with acceptable effort.

Generally speaking, any measures that make a representation more effective by making it easier to find and use the knowledge represented, also make the representation more efficient. For example, good search and selection algorithms or the possibility to provide different views help to increase efficiency as well as effectiveness.

An exception to this rule would be a measure that reduced user effort and time to find and use the knowledge, but that increased computer effort and time so much that the overall effort and time increased, e.g. due to particularly long searches.

explicitness: a representation scheme must be able to provide an explanation of its inferences.

While for inferred (structured) knowledge this can be taken literally, ill-structured knowledge is considered explicit if for any knowledge found, a more detailed description of this knowledge is available. Put differently, a representation notation is considered explicit if it allows different levels of detail or abstraction.

Example: a project leader has found in the process model that in order to create a project plan, he needs to analyze, among (many) other tasks, the risks involved in the project. The process model will now (upon request) show what needs to be done for a risk analysis, covering technical risks, financial risks, etc. The project leader might now "drill down" to find out more about typical financial risks and how to handle them.

In [36], the author listed a number of requirements on software process technology, some of which are relevant in the current context as well. These are requirements that have a focus on the industrial use of a process model.

accessibility: the representation must be accessible from different technical environments, and publication must be possible both in electronic and in paper format. This is less a requirement on the knowledge representation notation as such, but on the tool used to present the knowledge.

modifiability: the representation must support easy modifications of the process described, to support continuous improvement and adaptation to a changing environment (cf. Sec. 1.2 on meta-processes). Among other things, this implies that some kind of consistency checking must be available, to prevent modifications from introducing inconsistencies.

understandability: the representation must be easy to understand for developers. Their main objective is to understand the processes described and perform them. If developers have to think about the representation rather than the represented process, they will tend to ignore the defined process.

This requirement re-enforces the argument made above that knowledge representation notations for use by humans need to allow knowledge represented by free text, to make the more formal or structured knowledge easier to understand.

Search and Selection Mechanisms When describing software processes, one quickly runs into the problem that developers cannot find the description of a certain task when they need it. To overcome this problem, searching and selection must be provided as part of the software process support used. Typical mechanisms for this purpose are

- keyword search and full-text search
- different views based on various criteria such as role or project phase
- tailoring / only showing relevant detail
- intelligent agents [44, p. 59]
- visualization models [44, p. 60], such as Perspecta (`http://www.perspecta.com`) and InXight (`http://www.inxight.com`)
- Ellmer et. al. [25] describe a search strategy specifically suited for a process library, based on neural networks (self-organizing maps).

3.2. *Artificial Intelligence Approaches to Knowledge Representation*

Standard approaches in artificial intelligence to representing knowledge are predicate calculus, production rules, semantic networks, frames, and scripts [26]. A newer approach to structuring knowledge and finding it again once it has been captured in a knowledge base is the use of ontologies [45].

Analysis of these approaches quickly shows that all of them are quite suitable for structured knowledge to be used for automated reasoning, but do not support the representation of knowledge for use by human readers. In particular, they do not allow free text, and are therefore not sufficiently expressive for our purposes. On the other hand, dealing with pre- and post-conditions of tasks is clearly a strong point of AI notations.

Similarly, they tend to be fairly effective and efficient for structured knowledge (since they were designed for that purpose) but less effective and quite inefficient for human users, looking for ill-structured knowledge. Even for structured knowledge, however, they are not ideal since they cannot easily handle the different kinds of entities used in software process modeling, such as tasks, artifacts, roles, etc.

The explicitness of software process models in AI notation depends on the detailed approach and the tools used — no general answer can be given. Accessibility is typically low since the notation is supported by a single tool.

Modification of described processes is possible but not easy since it is difficult to understand the consequences of the modifications. More generally, it is difficult to understand a process model and its interactions if the model is described using AI notations. However, this is eased by the fact that these notations allow experimentation and asking questions about the model, such as "what are the possible steps (i.e. applicable rules) once I have created artifact X?".

3.3. *Software Process Modeling Approaches*

The following discussion is based on [24, 49, 55] which all contain an overview of software process modeling languages and approaches.

Software process modeling languages such as Appl/A [54], MSL (Marvel Strategy Language) [6, 12], and TEMPO [10, 11] are all very formal languages, used for automatically enacting or checking the processes, but not to be used by human users for their development tasks. Although SLANG [5] and Statemate [29, 33] are graphical representations of the software processes, they do not score much better regarding understandability. Only MVP-L (multi-view process modeling language) [16, 49] puts more emphasis on the human user and is therefore somewhat easier to understand — but still far from the kind of process model one would like to give to developers to tell them how they ought to perform a certain process.

These software process modeling languages are, to some extent, based upon ideas from AI knowledge representation, and therefore share many of their advantages and disadvantages. For example, they both are similarly effective and efficient for structured knowledge, and similarly ineffective and inefficient for unstructured

knowledge.

However, concepts such as tasks, artifacts and roles are fundamental to software process modeling languages and therefore well supported, improving the expressiveness for structured knowledge.

Explicitness of software process models as well as the ease with which they can be modified depend on the actual modeling language used.

3.4. *Business Process Modeling Approaches*

Since software processes are a special case of business processes, techniques for business process modeling can also be applied to software processes. Typically, business processes are modeled using some graphical, easy-to-understand representations such as flowcharts or Petri nets [1], or use cases and other UML notation [40], with attributes such as roles, effort, etc. included as hypertext. (See [19, p. 50] for further references.)

This approach can also be applied for software processes, although many of the attributes, such as the duration of a process step, often cannot usefully be filled in since software processes have less structure than typical business processes. As a result, many of the standard process analysis techniques cannot be applied.

When applying this approach to software processes in order to support developers, one has to make sure that lengthy free text explanations are possible — many tools only allow a few lines of explanation.

Since business process modeling languages are usually more geared towards analysis and simulation of the processes modeled, there is little support for the people actually performing the process, such as making software process knowledge available to them.

Since analysis and simulation is typically performed by a small number of people (the *Process Engineer* or the *Software Engineering Process Group*), there is little emphasis in the tools on making the model available within a heterogeneous environment, although sometimes a Web interface is provided.

Business process models tend to be quite explicit in that there are different levels of the model, with atomic tasks on one level being further broken down on the level below.

Modification of a business process model is quite easy, with various checks available to ensure that no inconsistency is introduced.

3.5. *Structured Plain Text*

Structured plain text is the approach to representing software process models that is used most often in practice, see for example the German V-Modell [56] or the *(Rational) Unified Process*, which additionally includes links to supporting tools, see [38] and `http://www.rational.com`.

Other common formats for representing software process models are HTML and document databases such as Lotus Notes. In particular, these formats score fairly

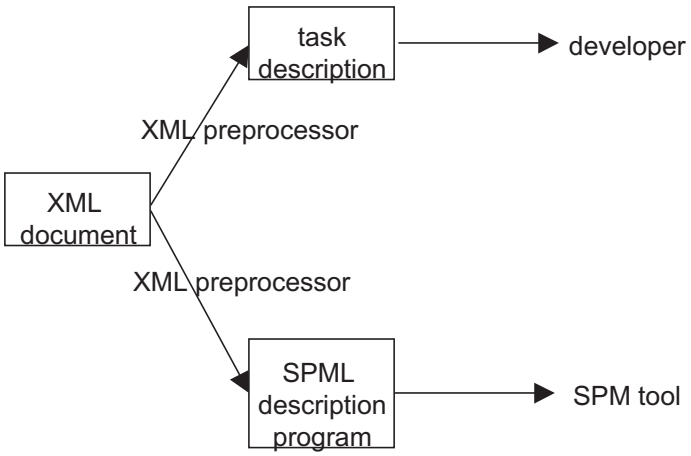

Fig. 4. Combining software process modeling languages (SPML)/tools with developer support using XML

high in accessibility, since they do not need special tools but use tools that many developers will have on their desk anyway, such as a browser.

XML A fairly recent approach to structure text is the use of the *Extensible Markup Language* XML [14]. XML was heavily influenced by SGML and HTML, allowing for more complex markup than HTML with individual tags, but at the same time being considerably less difficult to handle than SGML (see `http://www.w3.org/XML/`, `http://www.xml.com` and `http://www.oasis-open.org` for more information about XML).

XML allows different views on the same data, such as different levels (beginner, advanced, check list, detailed reference), for different user types (developers, automated tools), or for different project types (small or large projects; customer-specific or off-the-shelf software;)

To some extent, it is possible, using XML, to combine software process modeling approaches with additional descriptions in natural language that can be given to developers to help them perform their tasks. A pre-processor then has to remove these descriptions before the process model can be handled by the relevant enactment or analysis tools. Alternatively, the pre-processor removes the tool-specific parts to get a description of the software processes to be read and used by developers (Fig. 4). Some work in this direction is currently underway at the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern.

Process Patterns were first introduced by Coplien [22] and described more extensively by Ambler [2, 3, 4] (see also `http://www.ambysoft.com/processPatternsPage.html` for more resources on process patterns).

Each process pattern describes one problem in software development processes and suggests a solution to it. "A pattern is a general solution to a common problem or issue, one from which a specific solution may be derived" [2, p. 3].

Such a process pattern is described using the following structure:

- Problem
- Context
- Forces
- Solution
- Resulting context
- Design rationale (this item is included by Coplien but not by Ambler)

Ambler distinguishes process patterns at different levels of detail, namely task, stage and phase process patterns. Together, these process patterns can be used to describe a full process model. On the other hand, individual patterns are particularly suited to describing best practices.

Process patterns are easy to understand and modify, but there is no support ensuring that the model is still consistent.

Since process patterns define a structure for the definition of software processes but no technical format, an arbitrary (hyper-) text format can be used; as a result, it is easy to provide access to the process definition from whatever environment is needed.

As an approach to software process modeling, process patterns are quite different from XML (or Document Management Systems, see below), since they are concerned with the *logical* structure of process descriptions, while XML is concerned with their *physical* or technical structure. As a result, it is quite natural to combine process pattern with XML, where process patterns provide the chapter structure for process descriptions, and XML provides the syntax to denote the chapters and their contents, plus hypertext links as needed. A very simple example of a process pattern represented using XML is given in Fig. 5.

One of the strong points of XML is that it is easily accessible in a heterogeneous environment, including paper printout. Modification of XML documents is also easy, with some consistency checking possible (depending on the DTD and the tools used). Understanding an XML document itself, including the tags, is possible but not easy. However, one would not usually want to read the XML document itself, but only its presentation as defined by a style sheet. Assuming an appropriate style sheet, the presentation becomes easy to read.

Quality Patterns are quite similar to process patterns, but specific to solving quality-related problems and documenting quality-related experience. They

```
<?xml version="1.0" standalone="yes"?>
<processpattern>
<name>Form Follows Function</name>
<alias>Aggregate Roles into Activities</alias>
<problem>A project lacks well-defined roles.</problem>
<context>You know the key atomic process
activities.</context>
<forces>Activities are too small, and ...</forces>
<solution>Group closely related activities
...</solution>
<resultingcontext>A partial definition of roles for a
project.  ...</resultingcontext>
<designrationale>The quality of this pattern needs to
be reviewed.  ...</designrationale>
</processpattern>
```

Fig. 5. Example process pattern (from [22, Ch. 5]) using (simple) XML

were developed by Houdek et. al. [31, 32] and use a more elaborate pattern structure than process patterns, see Fig. 6. However, there is no obvious reason why this pattern structure should be specific to quality-related experience and not be used for process patterns in general.

Document Management Systems Software process models described as plain text can naturally be structured as a collection of documents. An obvious choice for storing such documents are document management systems (DMS) which support storage, retrieval and versioning of documents. A typical tool used here is Lotus Notes, although its support for versioning of documents is rather weak. Lotus Notes, like some other document management systems, additionally has groupware capabilities that can be used for discussions and feedback mechanisms on the process model.

DMS support representation of both structured and unstructured knowledge and are therefore quite expressive. However, retrieval of represented knowledge is usually only supported by search and selection algorithms, while inference of new knowledge from existing (structured) knowledge is not available.

A strong point of DMS is the administration of links between different knowledge items, making it easy to provide more detail as needed (explicitness).

Access to the knowledge represented using a DMS is only via this DMS, but most providers of DMS make an effort to provide versions that run on different platforms and can be used in an heterogeneous environment.

Modification of the knowledge represented in a DMS is easy, but it is also easy to introduce inconsistencies. Some of these inconsistencies can be found by the DMS (depending on the tool used), such as links leading nowhere, but

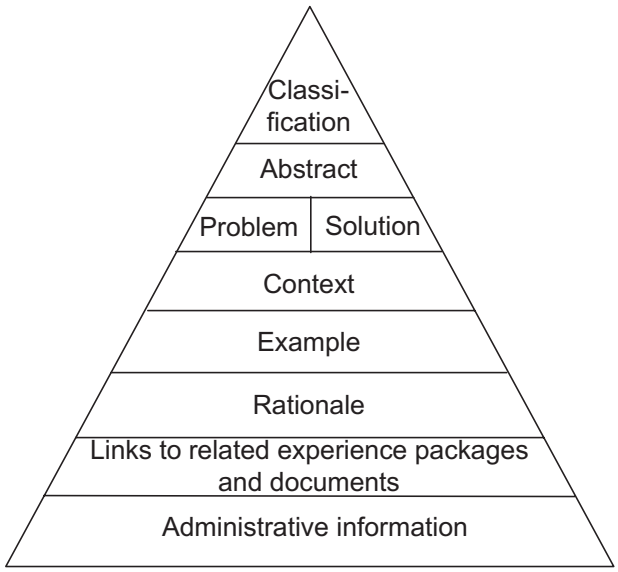

Fig. 6. Structure of a quality pattern [31]

inconsistencies within the texts cannot be found.

3.6. *Summary*

Table 1 gives an overview of the requirements on software process representation notations as described in Sec. 3.1, and how the notations listed in Sec. 3.2 to Sec. 3.5 satisfy these requirements.

For the requirements described here, the structured plain text approaches clearly seem most suitable, which was to be expected due to the requirement that the representation scheme must allow free text for the description of tasks etc.

In particular, XML looks promising as the way forward, since it supports the combined representation of free text and structured elements as needed for the automation of individual tasks. The challenge now lies in creating a document type definition (DTD) for process models that includes both aspects adequately. However, this is not an easy task since it amounts to defining a common meta-model for software process models in XML notation.

When talking about a *combined* representation, we really require that it is *one* integrated representation dealing with both aspects. Splitting the representation into two separate parts, containing a formal process model on the one hand and a human-readable description on the other, as suggested e. g. by [25], will lead to consistency problems, just like design documents hardly ever describe the design of the current implementation of a software system.

	AI approaches	software process modeling	business process modeling	structured plain text		
				XML	Process patterns	DMS
expressiveness:						
structured k.	0	+	+	+	−	0
unstructured k.	−	−	0/−	+	+	+
effectiveness:						
structured k.	+	+	0	+	−	−
unstructured k.	−	−	−	+	+	+
efficiency:						
structured k.	+	+	0	+	−	−
unstructured k.	−	−	−	+	0	+
explicitness	+/0/−	+/0/−	0/−	+	0	+
accessibility	−	−	+	+	+	0
modifiability	0	+/0/−	0/−	+/0	0	0
understandability	−	−	+	+	+	+

Table 1. Satisfaction of requirements on knowledge representation (+ fully satisfied; 0 partly satisfied; − not satisfied)

4. Managing Process Knowledge

As the term knowledge *management* suggests, handling knowledge is above all a management task rather than a technical problem of representing knowledge (even though some papers in this area, typically written by computer scientists, do give that impression). In this section, we will therefore look at the management aspects of process knowledge, in particular knowledge about software processes, and their implications on software process technology.

Well-known frameworks for the management of software process knowledge are the Experience Factory [7, 8, 9] and the Capability Maturity Model CMM [53], in particular the key process areas *Organizational Process Focus* (OPF) and *Organizational Process Definition* (OPD).

Additionally, most of the classic work [23] by Davenport and Prusak on knowledge management applies to the specific case of software process knowledge as well.

In the following, we will follow the structure defined by Probst (see Fig. 3).

4.1. *Knowledge Goals*

The main goal of knowledge management as described in the current article is to support developers in performing the software development processes, i.e. in developing software. This operational goal supports the strategic goal of getting faster, cheaper, and producing software of higher quality.

4.2. *Knowledge Identification*

The relevant knowledge is listed in Sec. 2.

4.3. *Knowledge Acquisition*

In order to represent knowledge in a process model, it must be provided by those who know, such as developers. To some extent, it can also be provided by a software process engineer, a member of the software engineering process group or similar, or even bought in from outside by using some standard model, but all these approaches are very limited if the process model is actually to be *used* by developers and not just put on the shelf.

To make developers put their development tasks aside and share their knowledge with others by helping to define the software process model, the organization must ensure that developers consider this as beneficial for themselves. In [23, Ch. 2], Davenport and Prusak describe three 'currencies' or potential benefits that can convince employees to share their knowledge:

reciprocity: "if I help to define the software processes, others will provide knowledge for the process model that I can use." To ensure this, the organization has to ensure that many qualified people contribute to the process model, not just a few people who do all the work and gain comparatively little benefit. Alternatively, the organization might provide explicit benefits such as time (not having to do other work while describing software processes) or even bonus payments or little extras such as a bottle of champagne.

repute: being known as the 'champion' or 'guru' for a certain topic. For software process technology, this implies that one should make clearly visible who contributed to a process model.

altruism: this does happen, but one cannot really plan on it for creating a process model.

If, on the other hand, providing knowledge for the process model is perceived as "keeping developers from doing real work", it just won't happen.

4.4. *Knowledge Development*

> "Knowledge development consists of all the management activities intended to produce new internal or external knowledge on both the individual and the collective level." [47, p. 24]

Typical activities to develop software process knowledge include

process improvement to refine and improve the knowledge about software processes.

packaging and representing knowledge once it has been created, turning implicit into explicit knowledge that is much better understood and ready for distribution.

training for those who define the processes, such as seminars in process modeling or attending conferences.

Training users such as developers in the defined processes, on the other hand, is part of knowledge distribution since it helps to spread the existing knowledge without creating any new knowledge.

4.5. *Knowledge Distribution*

Based on the representation of the knowledge described above, it has to be distributed to those who need it. Although in principle, such distribution can be on paper, this approach usually leads to write-only descriptions that is not used (shelf-ware). If one wants the knowledge to be not just distributed but to arrive in the people's heads, knowledge distribution must use channels such as

direct communication, using the various forms of training, such as standard taught seminars, training-on-the-job (assuming that it is performed as genuine training and not as a nice label on no training at all) or one-to-one coaching sessions. This in effect comes down to using the *personalization strategy* described by Hansen et. al. (see Sec. 1.4).

electronic distribution, making the description of the software processes available in an electronic format as described in Sec. 3 (applying Hansen's codification strategy).

4.6. *Knowledge Preservation*

Knowledge preservation deals with the fact that knowledge tends to disappear if nothing is done about it (knowledge entropy), because people forget or leave the company, or knowledge gets outdated.

To prevent such loss of knowledge, an adequate process for the preservation of the process (a meta-process) must be installed, for example as a continuous improvement process consisting of many small changes, or by defining releases of the process definitions on a longer time-frame.

4.7. *Knowledge Use*

Use of software process knowledge consists of developing and maintaining software, applying the knowledge contained in the defined development processes. Even if knowledge is developed and distributed, however, this is no guarantee that the knowledge will be used. To achieve this, developers need to be trained in the use of the defined processes, the processes need to be defined in a way such that the developers see an advantage for themselves using them, and finally quality assurance needs to check the use of the defined processes.

Although use of software process knowledge by an individual user is possible, this does not really gain a lot. To achieve its full benefit, the whole team needs to use and apply the knowledge and all work to the same processes.

4.8. *Knowledge Measurement*

Knowledge measurement is concerned with the evaluation and measurement of organizational knowledge. This measurement should be based on the goals that the organization wants to achieve with their knowledge management activities, in our case the support of developers in their task, and the faster, cheaper production of software of higher quality (cf. Sec. 4.1).

Example: one software house uses the following measurements to evaluate its organizational knowledge (this work is as yet unpublished):

number of improvement suggestions per month for the defined processes. Since no process definition is perfect, this is a good indicator of how intensively the knowledge contained in the process definitions is used and accepted.

number of read accesses to the individual parts of the process description. This is an indicator of which knowledge is used most and provides the most benefit.

At the same time, this is a good example of the possible problems with such measurements, since surprisingly, one of the documents accessed most often is out of date, as could already be seen from the title of the document. No good explanation of this effect was found.

number of deviations from the standard processes as identified by quality assurance reviews. This is a good indicator of the usefulness of the documented knowledge as seen by its users, as well as the effectiveness of the introduction of this knowledge.

5. Conclusions

As has been described in this paper, software process modeling for practical use can be viewed as a (knowledge) management task, not only a technical one. This has a number of implications for software process technology, such as

- emphasis on making software process models easy to use and accessible
- combination/integration of user-centred description and formal model. XML looks like a promising language for achieving this
- make visible who contributed what to a software process model
- software process knowledge should be presented on different levels of detail (checklist, detailed description, etc.)
- expressiveness, effectiveness, efficiency, explicitness, accessibility, modifiability, and understandability as described in Sec. 3.1.

There is already some work in the direction suggested here, see for example several papers at the SEKE'99 conference [37] under the heading of the *Learning Software*

Organization (see also the Chapter by Ruhe in this Handbook). Furthermore, most or even all the process models in widespread practical use, outside research institutions, follow the route suggested here and concentrate on support for the manual development tasks, typically by describing (in structured plain text) the tasks and artifacts and providing artifact templates. See e.g. the German V-Modell, the (Rational) Unified Process, internal, company-specific process model such as [43], the processes published on `http://sepo.spawar.navy.mil/docs.html` and on the Software Engineering Information Repository (SEIR) (`http://seir.sei.cmu.edu/`).

However, a lot of work still needs to be done before the results are satisfying for practical use. Currently, one can have either the benefits of the work done on software process technology (automated enactment of some process steps, consistency and other automated checks), or the kind of support for development tasks performed by humans as described in this paper. Hopefully, future work will be able to combine both.

References

1. W. van der Aalst, J. Desel, and A. Oberweis, Eds. *Business Process Management. Models, Techniques, and Empirical Studies.* Springer, 2000. LNCS 1806.
2. S.W. Ambler. An introduction to process patterns. White paper, AmbySoft Inc., 1998. Version of June 27, 1998.
   Online available at `http://www.ambysoft.com/processPatterns.pdf`.
3. S.W. Ambler. Process Patterns: Building Large-Scale Systems Using Object Technology. SIGS Books/Cambridge University Press, July 1998.
4. S.W. Ambler. More Process Patterns: Delivering Large-Scale Systems Using Object Technology. SIGS Books/Cambridge University Press, September 1998.
5. S.C. Bandinelli, A. Fuggetta, and S. Grigolli, Process Modeling in-the-large with SLANG in Proceedings of the Second International Conference on the Software Process (L. Osterweil, Ed.), pp. 75–83, IEEE Computer Society Press, February 1993.
6. N.S. Barghouti, D.S. Rosenblum, D.G. Belanger, and C. Alliegro, Two case studies in modeling real, corporate processes *Software Process — Improvement and Practice*, August 1995, pp. 17–32.
7. V. R. Basili, The Experience Factory and its Relationship to Other Improvement Paradigms in Software Engineering — ESEC'93, pp. 68–83, Springer LNCS 717, Heidelberg, 1993.
8. V. R. Basili, G. Caldiera, and H. D. Rombach, Experience Factory in Encyclopedia of Software Engineering Vol. 1 (J. Marciniak, Ed), pp. 469–476, Wiley, 1994.
9. V. R. Basili, F. Shull, and F. Lanubile Building Knowledge Through Families of Experiment *IEEE Transactions on Software Engineering*, August 1999, pp. 456–473.
10. N. Belkhatir, J. Estublier, and W.L. Melo, Software process model and work space control in the Adele system in Proceedings of the Second International Conference on the Software Process (L. Osterweil, Ed.), pp. 2–11, IEEE Computer Society Press, February 1993.
11. N. Belkhatir and W.L. Melo, Supporting software maintenance processes in TEMPO in Proceedings of the Conference on Software Maintenance, IEEE Computer Society Press, September 1993.
12. I. Ben-Shaul and G. Kaiser, A Paradigm for Decentralized Process Modeling, Kluwer,

Boston 1995.
13. B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.
14. N. Bradley, The XML Companion, Addison-Wesley, 1998.
15. G. Bremer, Genealogie von Entwicklungsschemata (in German) in Vorgehensmodelle für die betriebliche Anwendungsentwicklung (R. Kneuper, G. Müller-Luschnat, and A. Oberweis, Eds.), pp. 76–94. B.G. Teubner, Leipzig, 1998.
16. A. Bröckers, C.M. Lott, H.D. Rombach, and M. Verlage. MVP language report. Tech. Report 229/92, Dept. of Computer Science, Univ. of Kaiserslautern, Germany, 1992.
17. P. Brössler, Knowledge Management at a Software House. A Progress Report in Proc. of the Workshop on Learning Software Organizations, June 16, 1999, Kaiserslautern, Germany (Frank Bomarius, Ed), pp. 77–83, 1999.
18. K. Caputo. *CMM Implementation Guide. Choreographing Software Process Improvement.* Addison-Wesley, 1998.
19. Y-H. Chen-Burger, D. Robertson, and J. Stader. Formal Support for an Informal Business Modelling Method. *International Journal of Software Engineering and Knowledge Engineering*, 10(1):49–68, February 2000.
20. R. Conradi, A. Fuggetta, and M. L. Jaccheri, Six Theses on Software Process Research in Software Process Technology - 6^{th} European Workshop (EWSPT'98), Weybridge, UK(V. Gruhn, Ed), pp 100–104, Springer LNCS 1487, Heidelberg, 1998.
21. R. Conradi, Ed. Software Process Technology — 7^{th} European Workshop (EWSPT 2000), Kaprun, Austria. Springer, 2000. LNCS 1780.
22. J.O. Coplien. A generative development-process pattern language. In J.O. Coplien and D.C. Schmidt, Eds., Pattern Languages of Program Design. Addison-Wesley, 1995. Presented at PLoP/94, Monticello, Il., August 1994
    Online available at
    `http://www.bell-labs.com/people/cope/Patterns/Process/index.html`.
23. T.H. Davenport and L. Prusak, Working Knowledge. How Organizations Manage What They Know, Harvard Business School Press, Boston, 1998.
24. J.-C. Derniame, B.A. Kaba, and D. Wastell (Eds.), Software Process: Principles, Methodology and Technology Springer LNCS 1500, Heidelberg, 1999.
25. E. Ellmer, D. Merkl, G. Quirchmayr, and A. M. Tjoa, Process Model Reuse to Promote Organizational Learning in Software Development in Proc. of the 20th Annual Int. Computer Software and Applications Conference (COMPSAC'96), Seoul, South Korea, August 1996, pp. 21–26, IEEE CS Press, 1996.
26. J. Finlay and A. Dix, Artificial Intelligence, UCL Press, University College London, 1996.
27. V. Gruhn, Ed. Software Process Technology — 6^{th} European Workshop (EWSPT'98), Weybridge, UK. Springer, 1998. LNCS 1487.
28. M.T. Hansen, N. Nohria, and T. Tierney, What's Your Strategy for Managing Knowledge, *Harvard Business Review*, March 1999, pp. 106-116.
29. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, Statemate: a working environment for the development of complex reactive systems, *IEEE Transactions on Software Engineering*, April 1990.
30. J. Hertzberg. *Planen. Einführung in die Planerstellungsmethoden der Künstlichen Intelligenz.* Reihe Informatik, Band 65. BI Wissenschaftsverlag, 1989. (in German).
31. F. Houdek and C. Bunse, Transferring Experience: A Practical Approach and its Application on Software Inspections in Proc. of the Workshop on Learning Software Organizations, June 16, 1999, Kaiserslautern, Germany (Frank Bomarius, Ed), pp. 59–68, 1999.

32. F. Houdek and H. Kempter, Quality patterns — An approach to packaging software engineering experience in Proc. ACM Symposium on Software Reusability SSR'97, pp. 81–88, 1997.
33. W.S. Humphrey and M. Kellner, Software Process Modeling: Principles of entity process models in Proceedings of the Eleventh International Conference on Software Engineering, pp. 331–342, May 1989.
34. ISO/IEC, Information technology — Software life cycle processes ISO/IEC 12207, 1995.
35. ISO/IEC, Software Process Improvement and Capability dEtermination Standard ISO/IEC 15504, 1995.
36. R. Kneuper, Requirements on Software Process Technology from the Viewpoint of Commercial Software Development: Recommendations for Research Directions in Software Process Technology - 6^{th} European Workshop (EWSPT'98), Weybridge, UK(V. Gruhn, Ed), pp 111–115, Springer LNCS 1487, Heidelberg, 1998.
37. Knowledge Systems Institute,
SEKE'99. Proceedings. 11th Intern. Conf. on Software Engineering and Knowledge Engineering, 1999.
Online available at `http://www.ksi.edu/publication.html`).
38. P. Kruchten, The Rational Unified Process — An Introduction, Addison Wesley Longman, 1999.
39. T. S. Kuhn, The Structure of Scientific Revolutions, The University of Chicago Press, 2nd edition, 1970.
40. C. Marshall. *Enterprise Modeling with UML. Designing Successful Software Through Business Analysis.* Addison-Wesley, 1999.
41. B. McFeely. Ideal: A user's guide for software process improvement. Technical Report CMU/SEI-96-HB-001, Software Engineering Institute, Carnegie-Mellon University, 1996.
Online available at `http://www.sei.cmu.edu`.
42. J. Mylopoulos, A. Borgida, and E. Yu. Representing software engineering knowledge. *Automated Software Engineering*, 4:291–317, 1997.
43. J. Noack and B. Schienmann. Introducing OO development in a large banking organization. *IEEE Software*, pages 71–81, May 1999.
44. D.E. O'Leary, Enterprise Knowledge Management, *IEEE Computer*, March 1998, pp. 54–61.
45. D.E. O'Leary, Using AI in Knowledge Management: Knowledge Bases and Ontologies, *IEEE Intelligent Systems*, May/June 1998, pp. 34–39.
46. L. Osterweil, Software Processes are Software Too, in Proc. 9th International Conference on Software Engineering ICSE9, pp. 2–13, 1987.
47. G.J.B. Probst, Practical Knowledge Management: A Model That Works, *Prism*, Arthur D. Little, 2nd quarter 1998, pp. 17–29
Online available at `http://know.unige.ch/Prismartikel.pdf`.
48. I. Robertson, An Implementable Meta-process in Proceedings, Second World Congress on Integrated Design and Process Technology (M.M. Tanik, F.B. Bastani, D. Gibson, and P.J. Fielding, Ed), Society for Design and Process Science, 1996.
49. H.D. Rombach and M. Verlage, Directions in Software Process Research. *Advances in Computers* 41, 1995, pp. 1–63.
50. T. Rout, SPICE: A Framework for Software Process Assessment *Software Process — Improvement and Practice*, 1(1) 1995.
51. W. Royce. Why software costs so much. *IEEE Software*, 10(3):90–91, May 1993.
52. V. Schuppan and W. Rußwurm, A CMM-Based Evaluation of the V-Model 97 in Software Process Technology - 7^{th} European Workshop (EWSPT 2000), Kaprun, Austria

(R. Conradi, Ed), pp 69–83, Springer LNCS 1780, Heidelberg, 2000.
Online available at `http://www.inf.ethz.ch/ schuppan/paperewspt.pdf.zip`.

53. Software Engineering Institute, Carnegie Mellon University, The Capability Maturity Model. Guidelines for Improving the Software Process, The SEI Series in Software Engineering, Addison Wesley Longman, 1994.
54. R.N. Taylor, F.C. Belz, L.A. Clarke, L. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young Foundations for the Arcadia environment architecture in Proceedings of the Third ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments. Published as ACM SIGSOFT Software Engineering Notes (P. Henderson, Ed.), November 1988, pp. 1–13.
55. M. Verlage, Modellierungssprachen für Vorgehensmodelle (in German) in Vorgehensmodelle für die betriebliche Anwendungsentwicklung (R. Kneuper, G. Müller-Luschnat, and A. Oberweis, Ed), pp. 76–94. B.G. Teubner, Leipzig, 1998.
56. V-Model. Development Standard for IT-Systems of the Federal Republic of Germany. Lifecycle Process Model.
Online available at `http://www.v-modell.iabg.de/`.
57. J. Weizenbaum. Computer Power and Human Reason. From Judgment to Calculation. Freeman, 1978.