Using Case-Based Reasoning Technology to Build Learning Software Organizations

K.-D. Althoff, F. Bomarius, C. Tautz
Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
{althoff,bomarius,tautz}@iese.fhg.de

Abstract

Due to the ever increasing demands of the market, strategic management of knowledge assets, or *Learning Organizations* (LOs), are becoming a must in industrial software development. This paper presents work done at Fraunhofer IESE, where LOs for software development organizations are being developed and transferred into industrial practice. It describes how LOs for the software domain can be built upon both mature approaches from Software Engineering, like the *Quality Improvement Paradigm* (QIP) and *Experience Factory* Model (EF), and on industrial strength technology from AI, like *Case-Based Reasoning* (CBR). A system to support the *Learning Software Organization* (LSO) is sketched and experiences regarding the implementation of this system and LSOs in general are presented.

1 Introduction

The demands in today's software industry, such as short lead-time, frequent introduction of new technologies, increasing application complexity, and increasing quality requirements, are among the toughest to be found in industry. Traditional *production-oriented* approaches to meet these demands, like quality assurance or statistical process control, fall short or are just not applicable in the *development-oriented* software domain. In such an environment, continuous fast learning is one of the top priority requisites to acquire and maintain leading-edge competencies. Traditional individual or group learning, as a means of adapting to new demands or of adopting new methods and techniques, is often far too slow and ineffective. This is especially true if it is not pursued in a goal-oriented way, managed as a project crucial to a company's success, and supported by organizational, methodical, and technical means. So, learning on an organizational level and capitalizing on an organization's knowledge assets becomes imperative for modern software-dependent industries.

This situation is aggravated by the fact that the software engineering (SE) discipline has not yet evolved into a true *engineering* discipline in its short history. There is still much to be investigated and learned in applied SE, and such experience has to be thoroughly validated in industrial settings so as to come up with widely accepted procedures, techniques, and methods which eventually will comprise the core of a mature software engineering discipline. Only tight interaction of SE research with practice will significantly accelerate the maturation of the SE discipline, will bridge the gap between SE research and practice, and eventually will live up to meet the ever increasing demands the software industry is exposed to. In that endeavor, SE researchers as well as practitioners feel a strong need for powerful support in collecting experiences from industrial development projects, in packaging the experiences (e.g., build models from empirical data, formalize or semi-formalize informal knowledge), and in validating and spreading such *experience packages* into industrial projects.

The learning needs of both industry and research can be addressed by systematic application of Organizational Learning (OL) principles, supported by Organizational Memories (OM). The authors believe that Learning Organization principles will soon establish themselves as best practices. Therefore, we see a strong need to spell out OL procedures and

methods that work in practice and also a need for comprehensive tool support. This paper is about the approach taken by the authors to do so for the software domain.

The authors are affiliated with the Fraunhofer IESE, an institute with a mandate in applied research and technology transfer in SE. In its Experimental Software Engineering approach, the IESE employs learning cycles in all its operations (internally as well as in collaboration with customers).

From the IESE mandate the authors take on the viewpoints of applied research (collecting and packaging experiences from industrial development projects) and technology transfer (validating and spreading SE practices in industrial settings). We therefore see the subject matter as being composed of several dimensions:

- the processes, methods, techniques of how to implement OL in the application domain,
- the tools that support OL for that domain, and
- the organizational and cultural aspects of introduction and performance of OL.

From our experience we know that the latter one is of paramount importance for the success of a technology transfer project like, for instance, the introduction of OL [Kot96, Sen90]. However, we will not elaborate on this issue in the course of this paper.

In the following sections we first describe the approaches proposed and taken in the SE world towards continuous learning and improvement. This helps us give a definition of what we mean by learning and by OM in the software domain throughout this paper. We are then ready to derive requirements for a system to support OL in the SE domain. Next, we briefly explain why we consider CBR a good candidate technology and synthesize our vision of a system to support learning in the software domain. A brief look at example projects we are currently conducting, the agenda of our intended future work, and a conclusion end this paper. In the appendix we provide a comprehensive example for a learning cycle, to which we will refer when appropriate.

2 Approaches in the Software Engineering Domain

One of the fundamental premises of *Experimental Software Engineering* is that we wish to understand and improve software quality and productivity. Like in any engineering discipline, understanding and improvement must be based upon empirical evidence and all kinds of explicit (i.e., documented) project experiences. Even for small software organizations, large amounts of information could easily be accumulated over the years (e.g., project data, lessons learned, quality models, software artifacts, code databases). But, for such information to be usable, it needs to be modeled, structured, generalized, and stored in a reusable form in order to allow for effective retrieval of relevant (applicable) artifacts.

However, it is well known in the SE community that reuse does not happen easily across software development projects. This is not a surprise, since, by definition, the mission of a (traditional “pre-LO” style) development project is to deliver a product matching a given specification within given time and budget constraints. In other words, such a project is a strictly “local optimization” endeavor with a matching reward structure. Consequently, reuse, which by nature draws on results from *global* optimizations across projects, conflicts with the projects’ *local* goals. An individual’s extra efforts to enable future reuse of work results are not rewarded or are even penalized. Management slogans to “design for reuse” don’t work for that same reason.

In order to introduce LO mechanisms into such a culture for the first time, the continuous build-up and effective reuse of knowledge must be made into goals of OL projects which are separate from the development projects. Such learning- and reuse-related goals must be clearly defined, sufficient resources must be allocated, and the OL projects must be managed like any development project. In the short to mid-term, conducting such projects requires an organizational support structure that is separate from the software development organization. Once learning and reuse have become standard practices of software devel-

opment projects, most of the tasks and responsibilities of that support organization shall be assimilated by the regular development organization.

An organizational structure comprised of separate support and project development organizations has been proposed by Rombach and Basili and is called *Experience Factory Organization* [BCR94] (see Figure 1). The *Experience Factory* (EF) is an organization that supports software development projects conducted in what we call the *Project Organization* (e.g., a development department for a certain type of software) by analyzing and synthesizing all kinds of experiences drawn from these projects, acting as a repository for such experiences, and supplying those experiences back to projects on demand.

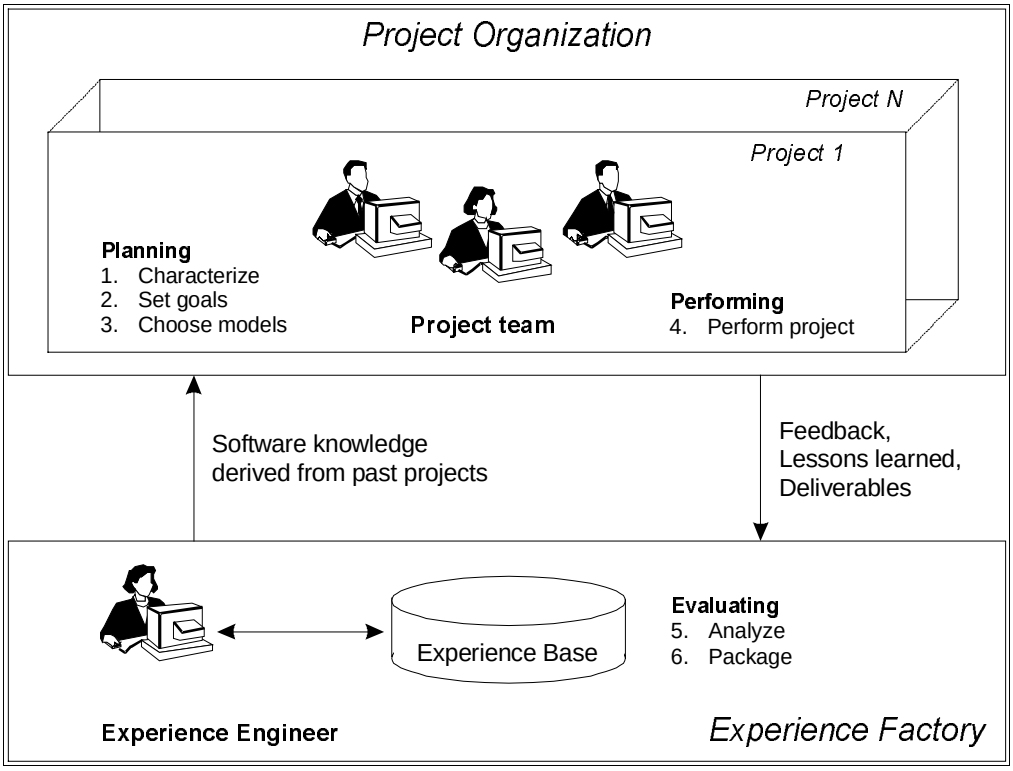

Figure 1: Mapping of QIP Steps into the Experience Factory Organization

The OL-related tasks of the experience factory and the project organization can be explained by mapping the steps of the *Quality Improvement Paradigm* (QIP) (see Figure 2) to the Experience Factory Organization (see Figure 1). The QIP is a generic six-step procedure for structuring software development and improvement activities. It involves three overall phases: planning, performing, and evaluating the software development project. The planning phase at the start of a new project is based on the explicit characterization (QIP1) of the initial situation, the identification of the project as well as learning and improvement goals to be achieved (QIP2), and the development of a suitable plan (QIP3) that shall achieve these goals. Steps QIP1, QIP2, and QIP 3 can benefit from the reuse of artifacts from the experience base (see appendix: “Getting the Project off the Ground”). The plan then guides the performance of the development project (QIP4), which is a feedback loop of its own controlling goal achievement. The subsequent evaluation phase involves the analysis of the performed actions (QIP5) and the packaging of the lessons learned into reusable artifacts (QIP6) (see appendix: “Learning from Project Experience”). The evaluation and packaging allows for effective reuse in similar projects in the future and may even set future learning goals (see appendix: “Strategic Aftermath”). They are the key means for making continuous improvement happen.

In summary, the six steps of the QIP are shown in Figure 2. In fact, the QIP implements two nested feedback/learning cycles: the project feedback/learning cycle (control cycle) and the corporate feedback/learning cycle (capitalization cycle). Project feedback is continuously provided to the project during the performance phase (QIP4). This requires empirical tech-

niques through which the actual course of the project can be monitored in order to identify and initiate corrective actions when needed. Corporate feedback is provided to the organization (i.e., other ongoing and future projects of the organization). It provides analytical information about project performance (e.g., cost and quality profiles) and accumulates other reusable experiences, such as software artifacts, lessons learned regarding tools, techniques and methods, etc., that are applicable to and useful for other projects. Throughout this paper we will focus on the capitalization cycle only.

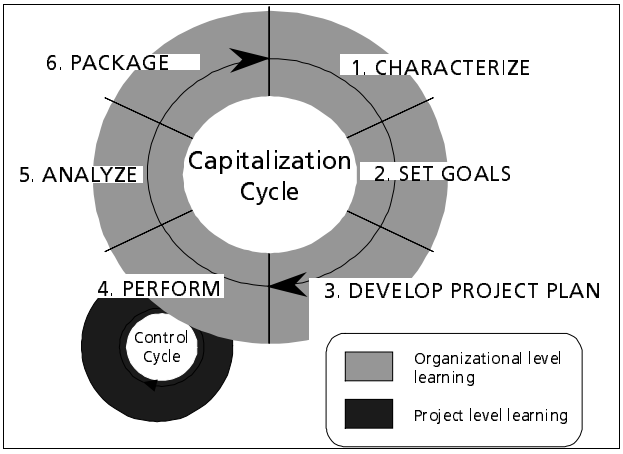

Figure 2: Quality Improvement Paradigm (QIP)

3 Requirements for a System to Support Learning in the Software Engineering Domain

In the context of this paper, *learning* refers to organizational learning rather than individual or team learning. Of course, our definition of OL will also support individual and team learning, but this is not the primary goal. Organizational learning under the needs specified in the introduction is targeted at accelerated elicitation and accumulation of explicit (documented) knowledge. Sources from which knowledge can be derived are experts (e.g., through interviews) as well as processes and artifacts (e.g., through measurement programs or experience reports). We place strong focus on explicit knowledge because the experts either take too long to build up the (tacit) knowledge a company would be interested in, or they simply become a scarce resource.

In the first case, a significant amount of diverse experiences would be needed on the part of the expert to build up knowledge about a given topic (for instance, to come up with a cost estimation model for a class of software projects). This usually means for the expert to take part in several projects in order to gather enough experience, which may take too long in a rapidly changing business environment.

In the latter case, they are often “eaten up” by coaching less experienced staff in more or less standard situations and therefore just cannot do “real” expert work. Making their knowledge explicitly available to the junior staff would help a lot. It is evident that we do not want to render experts obsolete. We rather strive to make them more valuable for the organization by relieving them of tutoring tasks and focusing them on the expert tasks.

In Figure 3 we give an abstract example of how the gathering of experiences and building of explicit knowledge takes place in Experimental Software Engineering. (Of course, a great deal of tacit expert knowledge will still be built up and remain in the minds of experts.)

Explicitly documented knowledge evolves along a timeline. On the “early” end of the line, a question or hypothesis raised by a business need (e.g., a quality problem) gives rise to look at measurement data (i.e., set up a measurement program) or interview experts. The first documented samples of information regarding the subject matter at hand are usually not directly transferable to other contexts, are often more a hypothesis than a fact, and often lack detailed information.

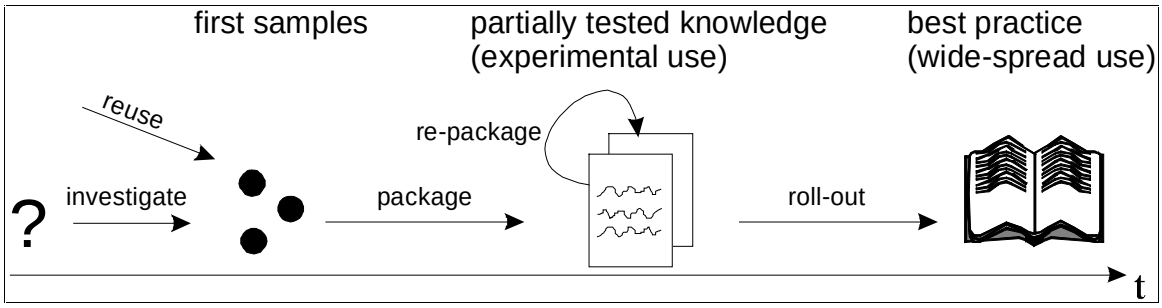

Figure 3: Experience Life Cycle

Over time, more samples regarding the question at hand are gathered and integrated (packaged). The information is described in a more structured form, new attributes are found to be relevant and information is possibly formalized (e.g., a mathematical model describing the quality problem is derived from data and expert opinion). Also, links to other documented experiences may be established or pointers to experts in that field are set.

A well-structured and easy to use representation of knowledge regarding the question we had in the beginning emerges and is made accessible to a wider circle of people. Its applicability is tested, it becomes more and more stable, and gradually evolves into a “best practice”.

In the long run, such proven practices become part of the standard operations and might be laid down in handbooks and manuals.

At any stage of the life cycle, information can be identified as being useless or no longer applicable. Such information will be forgotten. In technical terms this means to avoid creating data cemeteries.

From the abstract example above it is evident that we define an OM as the sum of explicit and implicit knowledge available in an organization. OL is in charge of creating, maintaining, and making available more and better explicit knowledge.

Figure 4 gives a brief example drawn from the more comprehensive scenario in the appendix for evolving artifacts, namely the retrieval, adaptation, and storage of the adapted process model.

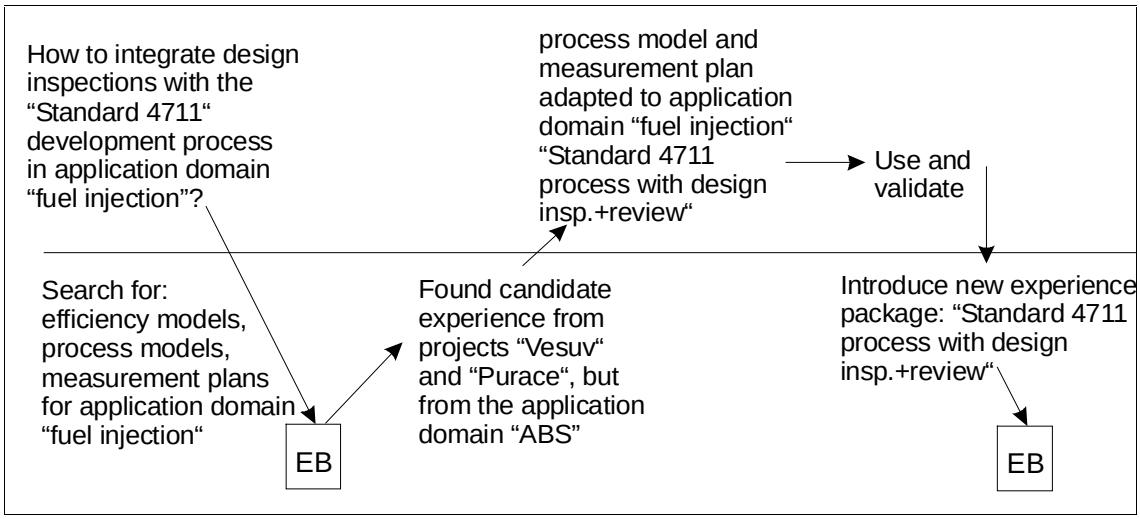

Figure 4: Example of Experience Life Cycle

In the following we list the requirements which have been found important for effective reuse of software engineering knowledge. The list of requirements is derived from [BCR94; BR91; Hen97] and from the practical experiences we gained in our projects with customers.

1. **Support for the mechanisms of incremental, continuous learning (according to QIP and EF)**. The elicitation of experiences from projects and the feedback of experience to the project organization requires functionality to support recording (collecting, qualifying, storing), packaging (specifying characterizations, tailoring, formalizing, generalizing, in-

tegrating), and effective reuse (identification, evaluation, selection, modification) of experiences as shown in Figure 5.

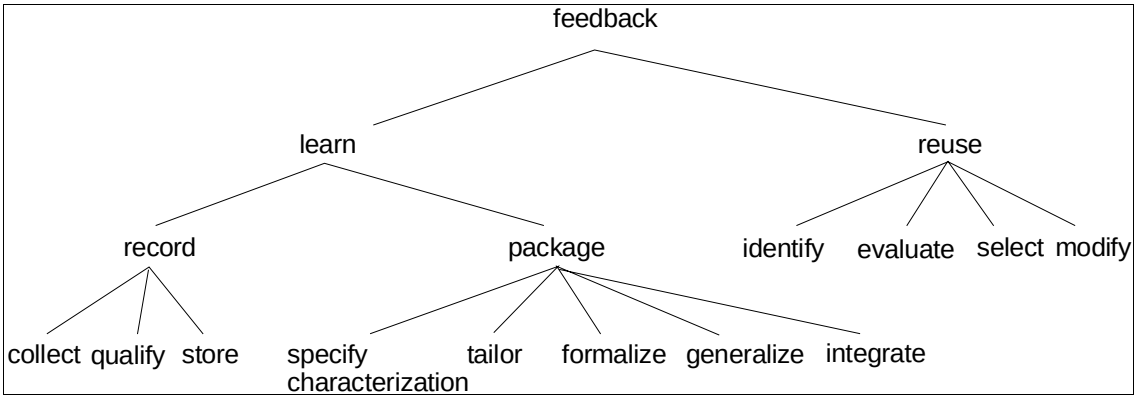

Figure 5: Mechanisms for incremental, continuous learning

2. **Storage of all kinds of software knowledge artifacts**. In the context of development and improvement programs, many kinds of artifacts need to be stored, such as process models, product models, resource models, quality models, all kinds of software artifacts (e.g., code modules, system documentation), lessons learned about these artifacts, and observations (results and success reports). All of these require characterization schemata of their own. Consequently, the OM will be integrated from a heterogeneous set of information sources. Integration of the diverse artifacts into a coherent model requires a flexible, modular overall schema for the OM that allows interfacing with various tools and information sources.
3. **Retrieval based on incomplete information**. Not all information needed for comprehensive characterization of an artifact is necessarily contained within the artifact or the OM at large. Nevertheless, an effective search of the OM must be possible. For instance, to reuse a code module, one would need to have:
   - information about the artifact itself (e.g., what language it is written in)
   - information about its interface (e.g., what is the functionality, what parameters and global variables need to be set)
   - information about its context (e.g., which application domain it was developed for, what life-cycle model was applied).

   If a software developer describes a code module he wants to search for, he may not have available or does not want to specify all of the above information. Therefore, it must be possible to search with incomplete information. See also the example in the appendix.
4. **Retrieval of similar artifacts**. Since each software development project is different, it is very unlikely that one finds an artifact exactly fulfilling one's needs. The retrieval mechanism must therefore be able to find similar artifacts, which may then be tailored to the specific needs of the project at hand.
5. **Support for maintenance of experience**. As an organization learns more about its software development operations, that knowledge needs to be reorganized. That means, an OM must be continuously maintained. For that purpose, the experience base must support the restructuring of software engineering knowledge. The decay of software engineering knowledge requires that experience already stored is easily maintainable in the sense that obsolete parts can be easily deleted or changed and remaining parts can be reorganized without changing the semantics of the experiences or creating inconsistencies.
6. **Learning from examples**. When software engineers encounter a problem, they think of past projects where similar problems occurred and adapt their solution to the current problem. Therefore, it must be possible to capture concrete examples as problem/solution pairs, i.e., a problem is specified using a characterization schema and the solution is the artifact itself. This is clearly an extension of "retrieval of similar artifacts".

7. **Support for continuously evaluating the quality and applicability of reuse artifacts and benefits of their application.** For identifying information eligible for storing in the OM, qualification and evaluation procedures and models are required on the "input side". On the "output side", the frequency of searching, applicability of artifacts found, ease of reuse/tailoring, and "user satisfaction" in general must be captured and used to control operations (i.e., goal setting for the learning organization, driving maintenance of the OM, etc.).

All these requirements are in line with what we have learned to be required in the learning and improvement programs conducted by the Fraunhofer IESE. Based on these requirements, we have evaluated candidate approaches to implement a system that supports OL in the software domain. Case-based reasoning technology is a promising approach [ABS96, Alt97a+b, TA97a] to us. In the following chapters we give a brief and informal rational why we consider CBR a good approach to help fulfil these requirements and then, how a tool suite can be built around a CBR system for comprehensive tool support.

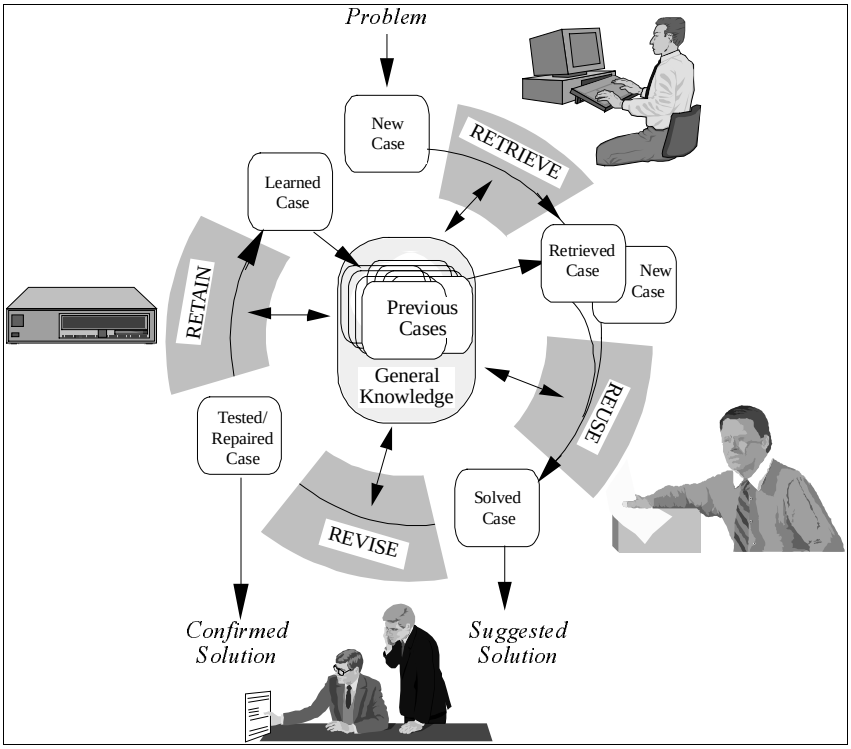

Figure 6: Organizational Level CBR: Human-based Case-Based Reasoning

4 Motivation for the Use of CBR Technology

There are two main arguments, a technical and an organizational one, why we selected CBR as the most promising AI technology to implement the above requirements:

- While the representation and reuse of software knowledge recommends an approach from the knowledge-based systems field, learning from examples suggests a machine learning approach. A technology that is rooted in both fields is CBR. It also offers a natural (i.e., direct) solution to the retrieval requirements.
- From an organizational perspective, the QIP has to be supported, which includes the mechanisms for incremental, sustained learning. [AW97] motivated that CBR can be applied on an organizational level. Here the basic CBR cycle consisting of the steps "retrieve – reuse – revise – retain", as described by Aamodt and Plaza [AP94], is completely carried out by humans (see Figure 6). In the following we want to call the application of CBR, which is independent from the technology employed, "Organizational Level CBR (OL-CBR)". [TA97a+b], [TA98], and [ABGT98] showed that QIP and OL-CBR are fully "compatible" (i.e., the respective cycle can be mapped to the other one in a se-

mantically reasonable way). In addition, [BA98] described how an experience factory organization is used to operationalize a methodology for developing CBR systems.

The application of CBR on the organizational level (i.e., OL-CBR) can be supported by various kinds of technologies (e.g., CBR-technology, knowledge-based system technology, WWW technology, database technology, etc.). For instance, a relational database can support the retrieval and retain steps (see Figure 6). However, for the first step, only restricted kinds of similarity assessment could be implemented using the relational database, and for the last step, learning issues would have to be ignored.

Regardless of the technologies employed, an analysis of the knowledge that is to be acquired, stored, and maintained, has to be performed. However, the degree of formality of the knowledge stored varies among the technologies.

Another aspect for the choice of the underlying technology is the stability of the knowledge. For the more stable parts of knowledge (i.e., knowledge that does not change very often, like a generic process model or a generic procedure for using a specific tool), more effort can be spent on its description and on reuse-enabling storage.

More dynamic knowledge (i.e., knowledge that changes often, and usually on a regular basis) is best described in a case-oriented way that is, dynamic knowledge is described using concrete cases that occurred in the past.

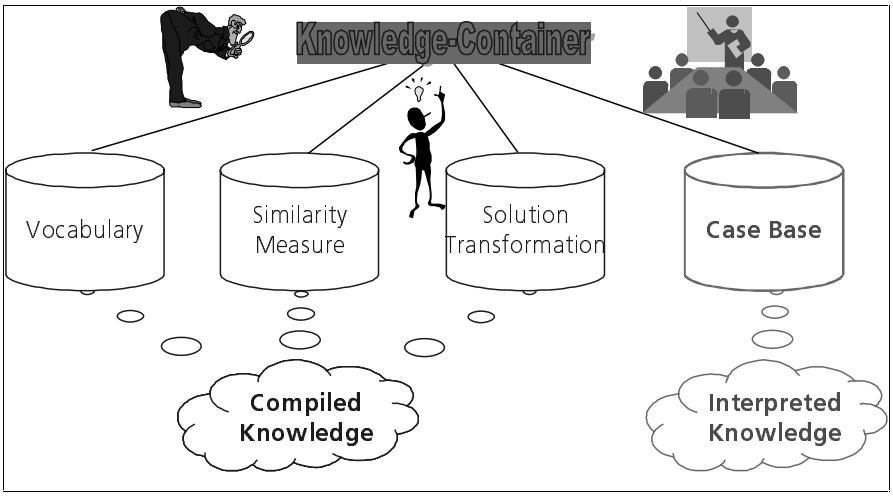

Figure 7: The knowledge container view on CBR systems (adapted from [Ric95])

All these considerations already make sense if cases are described informally (e.g., as lessons learned about a specific technology, or as results and success reports from an improvement program). It depends on the usage scenarios (i.e., on the customer's requirements) that shall be supported whether a semi-formal or formal representation is more appropriate. Usually, in the beginning the description is more informal and, over time, the degree of formality increases. In Figure 3 we illustrated this by means of the experience life cycle.

Richter [Ric95] has introduced a helpful explanation model or view on CBR systems, which supports our above arguments (see Figure 7). He identified four different knowledge containers for a CBR system. Besides the underlying vocabulary, these are the similarity measure, the solution transformation, and the cases. While the first three represent compiled knowledge (i.e., more stable knowledge is assigned to these containers), the cases are interpreted knowledge. As a consequence, newly added cases can be used directly. Therefore, the cases enable a CBR system to deal with dynamic knowledge more efficiently and effectively. In addition, knowledge can be shifted from one container to another when this knowledge has become more stable. For instance, in the beginning a simple vocabulary, a rough similarity measure, and no knowledge on solution transformation are used. However, a large number of cases are collected. Over time, the vocabulary can be refined and the similarity measure defined in higher accordance with the underlying domain. In addition, it may be possible to reduce the number of cases because the improved knowledge within

the other containers now enables the CBR system to better differentiate between the available cases. If knowledge about solution transformation also becomes available, it may be possible to further reduce the number of available cases, because then a number of solutions can be derived from one case.

To summarize this motivational section:

- We apply Organizational Level CBR (OL-CBR), which is compatible with the QIP and offers solutions for parts of the requirements in chapter 3 (e.g., Figure 3).
- We implement a set of support tools that are based on CBR (and other) technologies. They support OL-CBR in the SE domain and meet the requirements stated in chapter 3.

In the next section we describe the tools' architecture in some detail.

5 Synthesis of a System to Support Learning in the Software Engineering Domain

In this section, we present a system architecture for a "software engineering experience environment" (Figure 8). We distinguish between general purpose experience factory tools and application-specific tools. General purpose tools operate on the characterizations of the artifacts (attribute values which can be searched for) in the experience base, whereas the application tools operate on the artifacts themselves. Both kinds of tools act as clients using the experience base server as a means for retrieving and versioning software engineering artifacts. To illustrate the interplay of the different tools, we use the scenario described in the appendix.

During the QIP step 1 "characterize"[1], the general purpose search tool is started and the new project is (partially) specified guided by the characterization schemata (case models in terms of case-based reasoning tools) for project characterizations, techniques, and product models. The search tool then returns a list of similar project characterizations (cases in terms of case-based reasoning tools) by using the CBR tool of the experience base server. Starting from the project characterizations, the user can navigate to characterizations of relevant artifacts such as techniques employed, quantitative experiences collected, etc. The navigational search is supported through case references (links between cases), i.e., for each case the user navigates to, the CBR tool is used to retrieve it.

Next, goals are set (QIP step 2: "set goals") based on the results of the step "characterize". This is done in a similar manner: the search tool is invoked in order to find similar measurement goals from the past. However, these measurement goals and the respective measurement plans have to be adapted to project-specific needs. For this purpose, a cross-reference, which is stored as part of the artifact's characterization in the case base, is passed to the measurement planning tool. The measurement tool thus invoked loads the old measurement plan (using the data retrieval services of the experience base server) and allows the project manager to tailor it to the project's needs. The new measurement plan is saved using the data storage services, but its characterization is not specified yet. That is, no case is stored for it, and therefore it is not considered as part of the experience base. Thus it is not available to other projects (yet). This avoids the usage of unvalidated artifacts.

[1] The QIP step "characterize" should not be confused with the activity to specify the characterization of an artifact, because the purpose of this QIP step is to characterize the development environment as a whole and not a single artifact. In this report, we use the verb "characterize" to refer to the QIP step, while we use "specify the characterization" to enter into the experience base characteristic information about a single artifact.

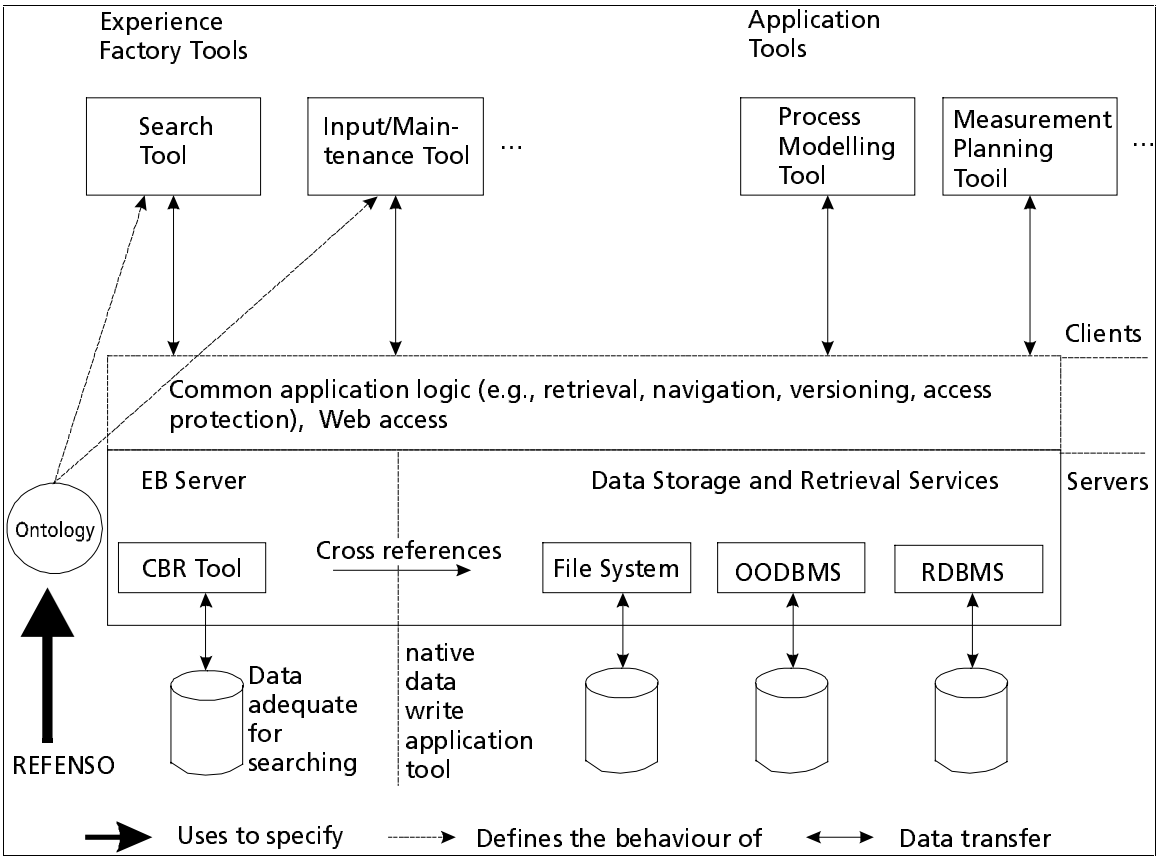

Figure 8: Architecture of the Software Engineering Experience Environment

In the same way, process models and other artifacts needed by the project are retrieved and tailored to specific needs of the project at hand (QIP step 3: "plan project"). The project is then executed (QIP step 4: "perform project").

Once the project is finished, the project's artifacts (which are deemed worthwhile to keep) are stored in the experience base for future projects (QIP step 5: "analyze"). For this purpose, the quality of the artifacts is determined through careful analysis with the help of the application tools. For those artifacts to be stored, the export-interface of the application tools compute the attributes' values of the attribute's characterization automatically as far as possible. This is necessary because the general purpose tools are not able to read the native data format of the application tools. Attribute values, which cannot be computed automatically, must be entered manually. This is realized through invoking the (general purpose) input and maintenance tool, which asks the user to supply the missing values. The tool determines the missing attribute values through retrieving the characterization scheme associated with the type of artifact (which is passed by the application tool as a parameter). This procedure is followed for all artifacts to be stored.

The newly inserted cases have to be validated and disseminated (QIP step 6: "package"). Therefore, they are initially marked as "to be validated" at the time of their insertion. Experience engineers from the experience factory assess the reuse potential of the newly acquired artifacts by invoking the respective application tool. At this point, the artifact may be modified to increase its reuse potential. Usually this requires modification of the artifact's characterization (using the maintenance tool). Finally, the corresponding cases are marked "validated".[2] After this, it is possible for other projects to access the new artifact.

In order to operationalize the retrieval, data input, and maintenance activities based on the characterization schemata, we need a formal representation of the "structure model" of an

[2] Statistical data can also be kept with the cases, so as to assess their usefulness (e.g., data on how often they had been applied successfully, or to what extent they had to be modified in order to be applicable).

experience base. The structure model can be seen in analogy to data models known from database management systems. In the AI literature, such structure models are known as ontologies [UG96]. For specifying software engineering ontologies, which aim at providing guidance for learning and retrieving software engineering knowledge, we use a special representation formalism named REFSENO (*re*presentation *f*ormalism for *s*oftware *en*gineering *o*ntologies) [TG98].

Currently, Fraunhofer IESE is implementing the architecture presented. We use commercially available software, namely CBR-Works from tecInno, Kaiserslautern [TEC98], as the CBR tool within the experience base server. The application and experience factory tools are built using the Java platform to ensure high portability across our customers' platforms.

6 Future Work

Fraunhofer IESE currently conducts a variety of projects within which OL/OM methods, techniques, and tools are further developed. Some are more research oriented, some have a clear mandate to improve an organization's performance. Here is a brief list of some of the projects and their main goals:

Health Care Sector	Develop the procedures, techniques, and tools for a KM system to comprehensively support the reorganization and optimization of hospitals according to the EFQM[3] framework.
Software Experience Center	In a consortium of multinational companies and know-how providers we develop the infrastructure for a 3-layer hierarchy of Experience Factories. The top level EF shall act as a cross-company EF and shall help member companies to effectively and efficiently exchange experiences on building EFs and conducting improvement programs.
Support for Tailoring SE Technologies	Over time we want to document explicitly our competencies regarding the SE technologies we transfer into practice. This documentation is based on feedback from organizations, which apply the technologies (e.g., lessons learned). The aim of this project is to build up knowledge on when to apply which variant of a given SE technology.
Decision Support for Tool Selection	This project has been an exercise for learning how to use CBR technology for tool selection, and also for developing appropriate evaluation criteria. We used the CBR-Works system from tecinno GmbH. A more detailed description is given in [ANT98]. This system is accessible at URL: http://www.iese.fhg.de/Competences/QPE/QE/CBR-PEB.html.
Several EF-related Projects	In these projects we build EFs for customer companies. Often the topic area from which experiences are drawn is narrow and the primary focus is on improving software development. An EF is built up as a means to capture experiences made and to slowly introduce and get accustomed to OL-style working.

In the context of ongoing and future projects we will refine our understanding of the requirements and mechanisms underlying the processes that enact OL. We will investigate alternative approaches and techniques to capture and model knowledge from the SE domain in formal, semi-formal as well as informal ways. We need to better understand the tradeoffs associated with the alternatives so as to give guidance not only on technical and organizational issues, but also in terms of economics. And we have to exercise our ideas in different organizational "cultures", so as to make them widely applicable and also understand the "soft" factors that are prerequisites to successfully implementing OL.

[3] The European Foundation for Quality Management. Awards the European Quality Award (EQA).

We are closely collaborating with the provider of the CBR system CBR-Works so as to extend the CBR tool towards supporting requirements from our system. In 1998 we plan to implement major parts of the proposed architecture. As of today, CBR-Works is a commercially available tool and most of the other tools that extend CBR-Works or that will be linked into the system already exist and are about to be deployed.

As a first full-fledged implementation, we are building the IESE Experience Factory, a system to store and make available IESE-internally the experiences from all our projects. This will allow us to gain valuable insights into how to actually install and make use of such a system. It will also allow us to continuously improve the system. In fact, in projects with industry we experience a tremendous request to implement OL in their environments, including the use of our system, even if it is only in a prototypical state.

7 Conclusion

Our work towards developing the Learning Software Organization has clearly shown that the core operations around the OM must be organized in a separate organizational unit. This corresponds to the separation into the "project organization" and the "experience factory" as put forward in chapter 2. Such an experience factory has its own responsibilities, conducts its own projects, is governed by goals that are linked to the business goals of the overall organization, has its own resources, and requires personnel with particular skills that are different from the skills needed in project organization.[4]

This organizational separation mentioned above is a prerequisite for OL to happen because

- the EF tasks must be protected from being "eaten up" as soon as development projects come under pressure, and because
- the EF pursues goals that are in part conflicting with the goals of the development projects – we discussed global vs. local optimization in chapter 2.

On the other hand, and this reveals an inherent conflict, only very close collaboration between the experience factory and the development projects will result in a lively learning organization where both parts, and thus the whole organization, benefit.

We have learned that successful integration is a cultural problem, rather than a technical problem. In most of today's software development organizations we find a culture where learning from past experience is strongly discouraged or even penalized because it requires to admit to mistakes. And mistakes are not considered a chance for learning and improving, but rather a reason to blame people. This clearly dominates the technical problems we are facing in our daily work.

From the technical point of view we have learned that a system to support LO must be an open system. We cannot expect a closed system to solve problems in industrial strength environments with a diversity of tools, methods, application domains as those found in software engineering at large. Integration of various tools wrt data and functionality is considered mandatory. This is reflected in our architecture. The expert system tool we use is one tool among others; however, it plays a central role. It ties together the diverse experiences and makes them effectively and efficiently accessible and eventually enables learning. This approach also avoids the plateau effect, widely known from "classical" expert systems.

The experiences stored in the OM have to be continuously worked on. As new cases are added, network complexity increases, new attributes and new case types are added. In order to tame complexity and sustain easy usability of the whole system, the OM has to be cleaned up on a regular basis. We consider this a crucial task of the experience factory. It has to be driven by empirical evidence that is continuously gained from the evaluation of

[4] Therefore we consider the LSO to actually be a knowledge management (KM) system. From our point of view there is no question whether a KM system adds value – KM is mandatory to make OL a professional endeavor.

effectiveness and efficiency of storage, retrieval, and reuse. Requirement 7 is therefore a very crucial one that must not be underestimated.

8 Acknowledgments

The authors would like to thank the workshop paper reviewers and Sonnhild Namingha for reviewing the paper, Susanne Hartkopf and Andreas Birk who contributed to an early version of this paper, Stefan Wess from tecinno GmbH, and many IESE employees for fruitful discussions.

9 References

[AABM95] K.-D. Althoff, E. Auriol, R. Barletta, and M. Manago. *A Review of Industrial Case-Based Reasoning Tools*. AI Intelligence. Oxford (UK), 1995.

[ABGT98] K.-D. Althoff, A. Birk, C. Gresse von Wangenheim, and C. Tautz. *Case-Based Reasoning for Experimental Software Engineering*. To appear in: M. Lenz, B. Bartsch-Spörl, H. D. Burkhard & S. Wess, editors, Case-Based Reasoning Technology – From Foundations to Applications. Springer Verlag,1998.

[ABS96] Klaus-Dieter Althoff and Brigitte Bartsch-Spörl. *Decision support for case-based applications*. Wirtschaftsinformatik, 38(1):8–16, February 1996.

[Alt97a] K.-D. Althoff. *Evaluating Case-Based Reasoning Systems: The Inreca Case Study*. Postdoctoral Thesis (Habilitationschrift), Department of Computer Science, University of Kaiserslautern, July 1997.

[Alt97b] K.-D. Althoff. *Validating Case-Based Reasoning Systems*. In: W. S. Wittig and G. Grieser (eds.), Proc. 5. Leipziger Informatik-Tage, Forschungsinstitut für Informationstechnologien e.V., Leipzig,1997, 157-168.

[ANT98] K.-D. Althoff, M. Nick and C. Tautz. Concepts for Reuse in the Experience Factory and Their Implementation for Case-Based Reasoning System Development. Technical Report IESE-Report, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern (Germany), 1998.

[AP94] Agnar Aamodt and Enric Plaza. *Case-based reasoning: Foundational issues, methodological variations, and system approaches*. AICOM, 7(1):39–59, March 1994.

[AW97] Klaus-Dieter Althoff and Wolfgang Wilke. *Potential uses of case-based reasoning in experience based construction of software systems and business process support*. In R. Bergmann and W. Wilke, editors, Proceedings of the 5th German Workshop on Case-Based Reasoning, LSA-97-01E, pages 31–38. Centre for Learning Systems and Applications, University of Kaiserslautern, March 1997.

[BA98] R. Bergmann and K.-D. Althoff (1998). Methodology for Building Case-Based Reasoning Applications. In: M. Lenz, B. Bartsch-Spörl, H. D. Burkhard & S. Wess (eds.), *Case-Based Reasoning Technology – From Foundations to Applications*. Springer Verlag, LNAI 1400, 299-328.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. *Experience Factory*. In John J. Marciniak, editor, Encyclopedia of Software Engineering, volume 1, pages 469–476. John Wiley & Sons, 1994.

[BR91] Victor R. Basili and H. Dieter Rombach. *Support for comprehensive reuse*. IEEE Software Engineering Journal, 6(5):303–316, September 1991.

[Hen97] Scott Henninger. *Capturing and Formalizing Best Practices in a Software Development Organization*. In Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, Madrid, Spain, June 1997.

[Kot96] John P. Kotter. *Leading Change*. Harvard Business School Press, Boston, 1996.

[Ric95] Michael M. Richter. The Knowledge Contained in Similarity Measures. Invited talk at the First International Conference on CBR 1995 (ICCBR-95). Slide copies and abstract available via *URL: http://wwwagr.informatik.uni-kl.de/~lsa/CBR/Richtericcbr95remarks.html*

[Sen90] P.M. Senge. *The fifth discipline: The art and practice of the learning organization*. Doubleday Currency, New York, 1990

[TA97a] Carsten Tautz and Klaus-Dieter Althoff. *Using case-based reasoning for reusing software knowledge*. In Proceedings of the 2nd International Conference on Case-Based Reasoning, Providence, RI, July 1997. Springer-Verlag.

[TA97b] Carsten Tautz and Klaus-Dieter Althoff. *Operationalizing the Reuse of Software Knowledge Using Case-Based Reasoning*. Technical Report IESE-Report No. 017.97/E, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern (Germany), 1997.

[TA98] Carsten Tautz and Klaus-Dieter Althoff. Operationalizing Comprehensive Software Knowledge Reuse Based on CBR Methods. In L. Gierl & M. Lenz (eds.), Proc. of the 6th German Workshop on Case-Based Reasoning (GWCBR-98), Technical Report, University of Rostock, IMIB series vol. 7, March 1998, 89-98.

[TEC98] CBR-Works (http://www.tecinno.de/tecinno_e/ecbrwork.htm)

[TG98] Carsten Tautz and Christiane Gresse von Wangenheim. REFSENO: A representation formalism for software engineering ontologies. Technical Report IESE-Report No. 015.98/E, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern (Germany), 1998.

[UG96] Mike Uschold and Michael Gruninger. Ontologies: Principles, methods, and applications. *The Knowledge Engineering Review*, 11(2): 93-136, 1996.

10 Appendix – Sample Scenario

This scenario shows how a:

- software development project is planned and performed using experiences from past projects provided by the experience base
- software development organization learns, i.e., how the contents of the experience base are enhanced and restructured according to new project experience.

10.1 Simplified Structure Model for the Experience Base

Before the mechanisms for reusing and learning software engineering knowledge can be explained, the structure model of the experience base must be presented. The structure model of the experience base can be seen in analogy to the data models of database management systems. It guides the user while he/she retrieves or stores knowledge.

The experience base shall contain several types of knowledge each represented by a separate case model. Every case in the case base is described by exactly one of these case models. During retrieval, one of the case models is used as a template to be filled in, in order to specify the knowledge to be searched for. This implies that the user has to know the type of knowledge he needs when he formulates a query. The type of knowledge can be regarded as a filter: only cases described by the selected case model will be returned.

For the scenario described here, the structure model shown in Figure 9 is used. The meaning of the different case models is described in the Table below. Each case model has two kinds of attributes: terminal and nonterminal attributes. Terminal attributes model how software engineering entities are specified for storage and retrieval, whereas nonterminal attributes model semantic relationships. Nonterminal attributes are implemented using references. All semantic relationships are bidirectional. This is indicated in Figure 9 by the arcs between the case models.

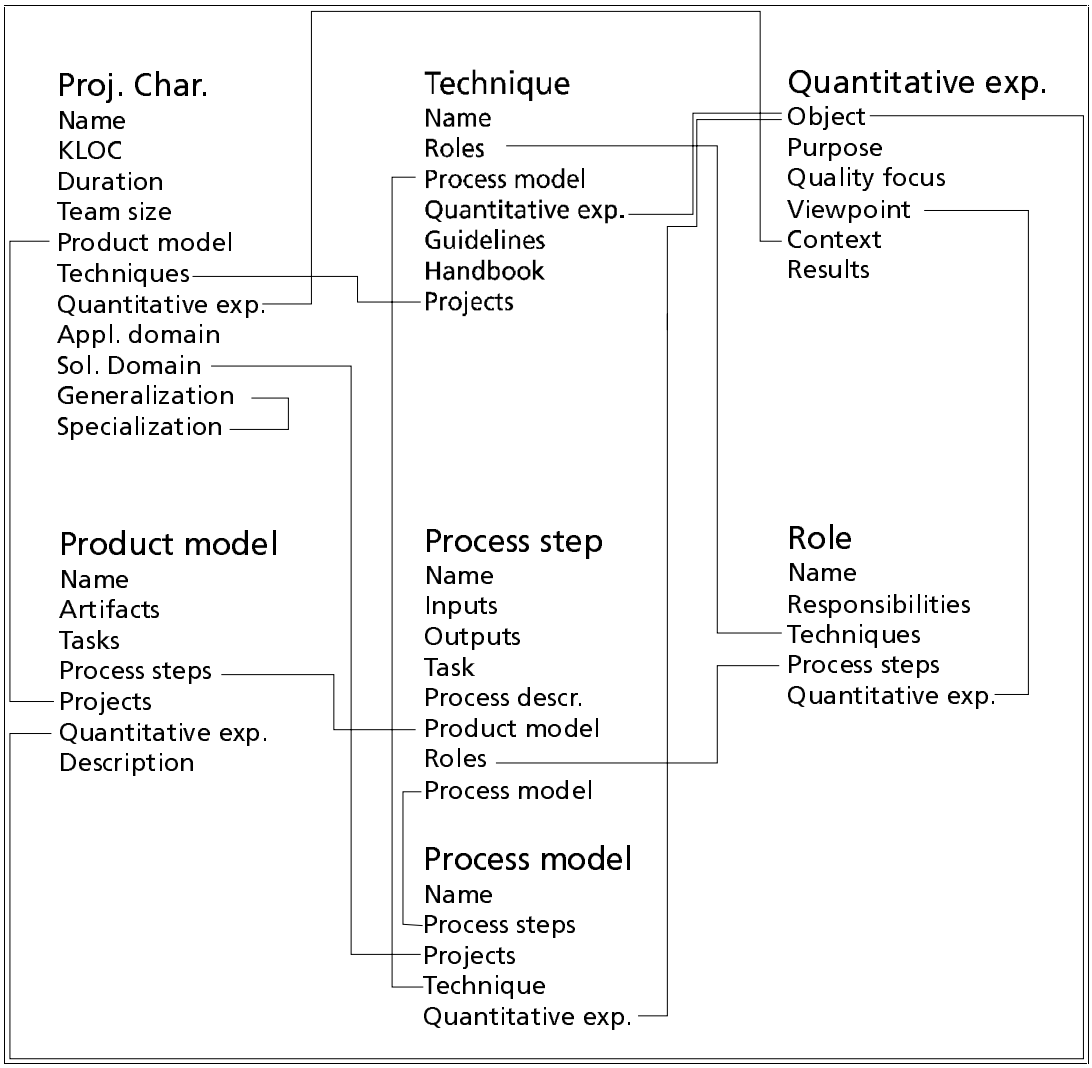

Figure 9: Simplified structure model of an exemplary experience base

Process Model	A process model specifies in which order which process steps are performed
Process Step	A process step is an atomic action of a process that has no externally visible substructure.
Product Model	A product model defines the structure of software development products as well as the tasks to be performed. It does not describe, however, how to perform these tasks (described by the corresponding process step) nor in which order the tasks are to be performed (described by the process model)
Project Characterization	A project characterization summarizes the relevant characteristics of a project. It contains applicability conditions for most other types of software engineering knowledge.
Quantitative Experience	Quantitative experience is a pair consisting of a measurement goal and the results of the measurement. It is collected in a goal-oriented way. This means that the measurement goal is defined at the beginning of a project. Such a goal is described using five facets: the object to be analyzed, the purpose for measuring, the property to be measured (quality focus), the role for which the data is collected and interpreted (viewpoint), and the context in which the data is collected. The data collected and interpreted is only valid within the specified context.
Role	A role is defined through a set of responsibilities and enacted by humans.

Technique	A technique is a prescription of how to represent a software development product and/or a basic algorithm or set of steps to be followed in constructing or assessing a software development product.

10.2 Project Setting

The fictive scenario described below is based on the following assumptions:

- the experience factory is established at department level at an automotive equipment manufacturer
- the department is made up of several groups; however, in the scenario only the groups responsible for the software development of "ABS" and "fuel injection" equipment are involved
- the group "fuel injection" has just closed a contract with a car maker requiring a "design review", something the group has never done before
- the new project is named "Maui"

10.3 Getting the Project off the Ground

10.3.1 Characterize

The project manager has never managed a project with a design review before. Therefore, he needs information on projects conducted where a design review has been performed. From the requirement to perform a "design review" he deduces that the software documentation must at least contain the requirements and the design. Furthermore, the product model must allow the construction and verification of these products (according to the glossary of the organization, the "design review" is a verification of the design). He estimates that the project will run 12 months with 3-5 people working on it at any given time.

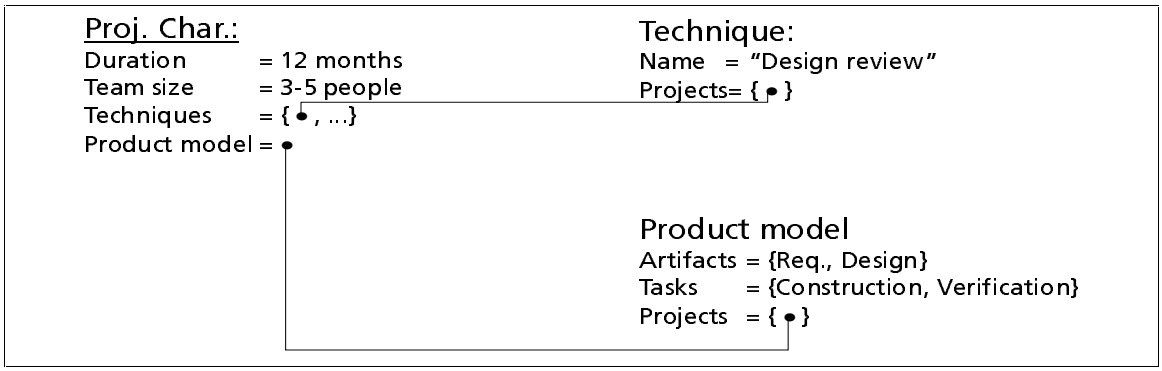

Figure 10: Query for similar project characterizations

As a first step the project manager enters his knowledge in the form of a query searching for similar projects (in our scenario these are projects with roughly the same duration and team size) which also employed design reviews (Figure 10). The three most promising project characterizations returned by the Experience Base are shown in Figure 11. As can be seen, two projects, named "Hugo" and "Judy", have been performed using design reviews.

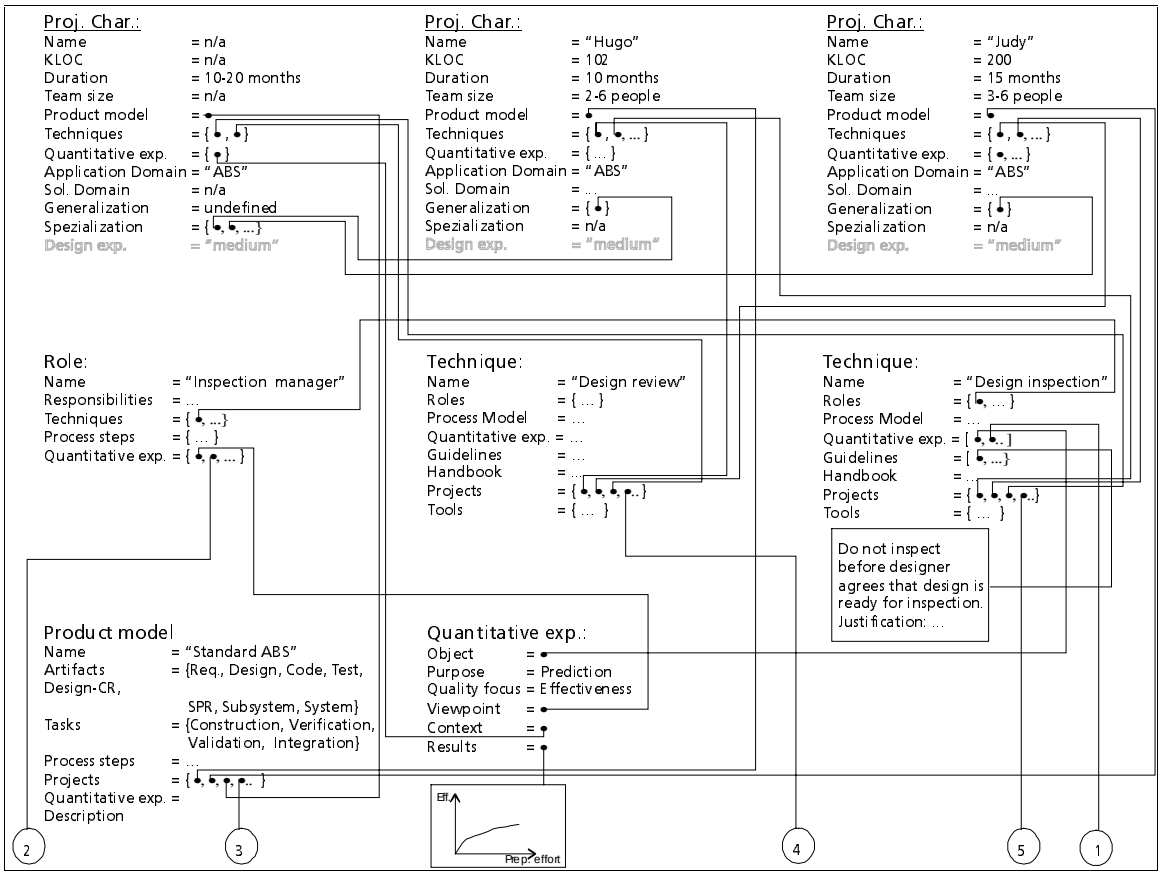

Figure 11: Result of first query

However, they have not been performed in the "fuel injection" group, but rather in the "ABS" group. Quite strikingly, in both cases a design inspection was performed besides the design review. A project characterization generalizing the characterizations of "Hugo" and "Judy" shows this explicitly.

By interviewing the project manager of "Judy", our project manager finds out that the inspections were performed for preparing the design review. The goal was to identify and eliminate as many design defects as possible before the customer takes a look at the design, thus increasing the confidence and/or satisfaction of the customer. Based on this discussion, our project manager decides to employ design inspections as well.

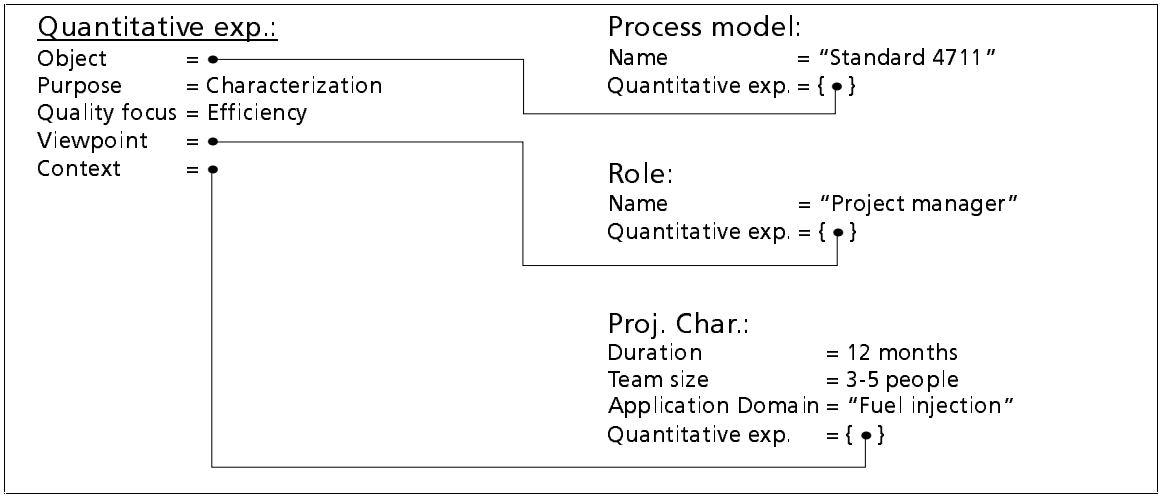

Figure 12: Query for similar quantitative experience

10.3.2 Set Goals

As the experience about inspections stems from a different application domain (i.e., ABS system development), the models available may not be valid for the application domain at hand. Therefore, it is decided (i.e., a measurement goal is set) to measure the effectiveness of inspections in this application domain, so as to extend the experience base by effectiveness models for inspections in the “fuel-injection” domain.

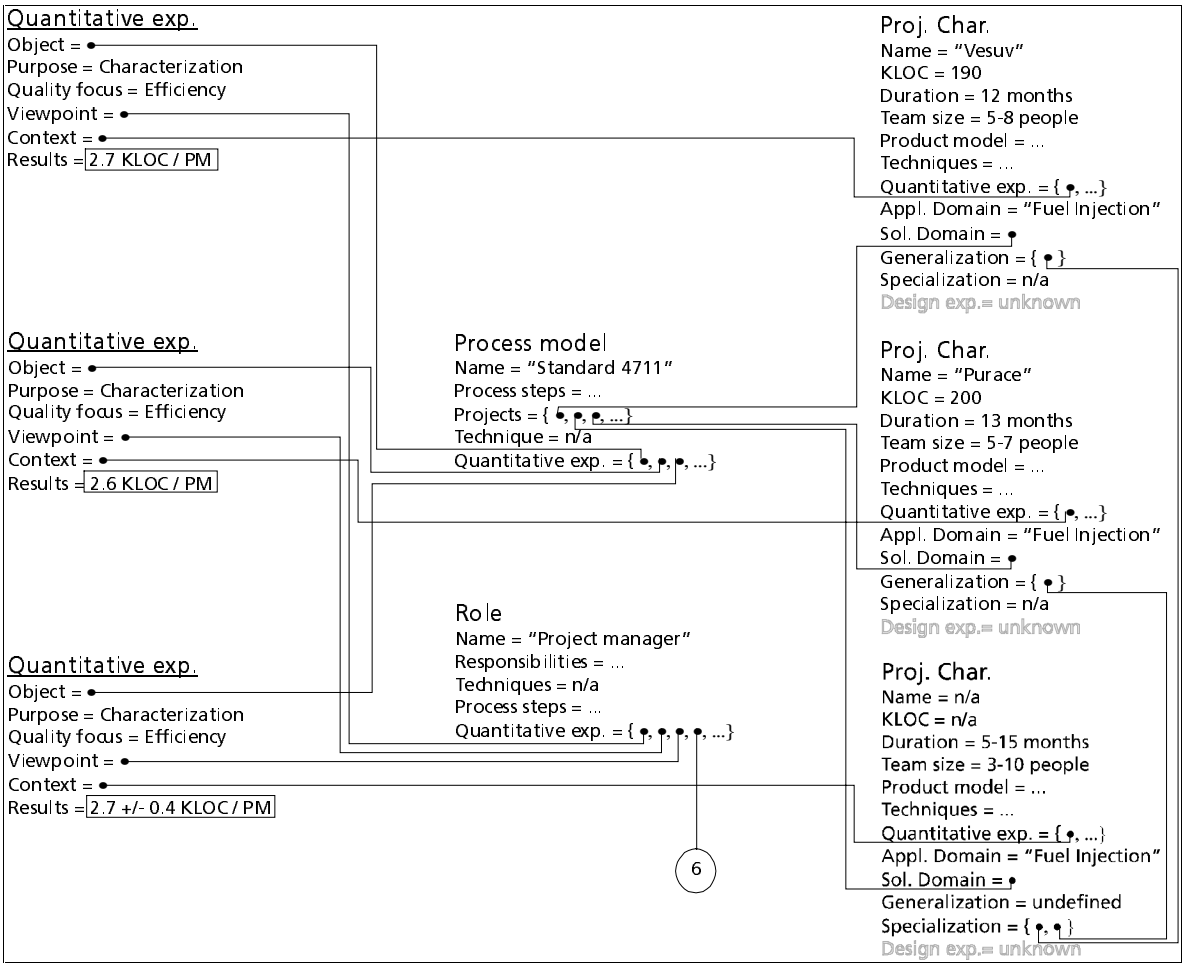

Figure 13: Result of second query

Furthermore, inspections are seen as a chance to improve the overall efficiency of software projects, because defects can be found in the life cycle earlier than can be if system tests were conducted at the end of coding and integration. It is therefore hypothesized that the rework effort for the correction of defects can be greatly reduced. In order to validate this hypothesis with quantitative data, a second query is formulated (Figure 12). The query searches for quantitative experiences on efficiency which were collected on similar projects in the "fuel injection" group using the standard process model "Standard 4711" which is to be used in "Maui".

The results of the query (Figure 13) show an efficiency range of 2.7+-0.4 KLOC per person month. If inspections make projects more efficient, the efficiency of "Maui" should be higher than 3.1 KLOC/PM.

10.3.3 Develop Project Plan

As the final planning step for “Maui”, the actual process models and measurement plans are being developed. The process model "Standard 4711" is taken as a basis and extended by design inspections and reviews. This results in a new process model "Standard 4711 with design insp.+review". The measurement plan used in the old projects "Vesuv" and "Purace"

is tailored to the new needs, i.e., the effort for performing the inspections is also considered for the computation of the efficiency.

In order to plan the inspections, our project manager also relies on the quantitative experience gained in the group "ABS". For example, he sets the goal to achieve an effectiveness of 0.5 and estimates the needed preparation effort based on this goal.

At the same time he identifies this as a risk factor, since the model upon which these estimations are based has not been validated for "fuel injection" projects (Subsection 10.3.2).

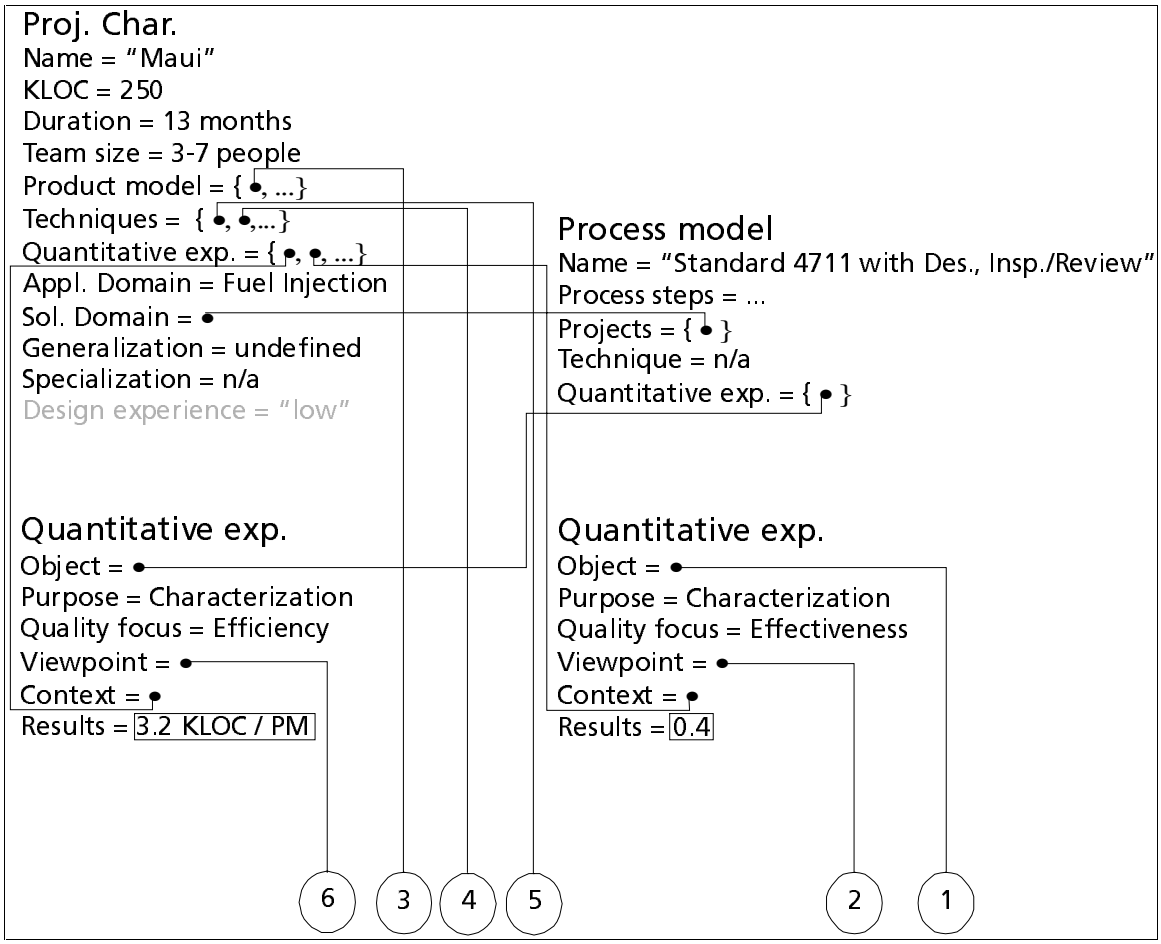

Figure 14: Updated experience base

Looking for further risk factors, our project manager also searches the experience base for guidelines associated with the techniques applied. For instance, the guideline "Do not inspect before the designer agrees that the design is ready for inspection" was found and will be employed because the justification sounds reasonable.

10.4 Perform Project

During the performance of the project, the data for the defined measurement goals is collected. In addition, the experience base is consulted for reusable components and problem solution statements.

Detailing these reuse attempts is, however, beyond the scope of this scenario.

10.5 Learning from Project Experience

10.5.1 Analyze

After the project was completed, 250 KLOC had been developed. Instead of the planned maximum of 5 people, 7 people had been working on the project. Also, the project duration was prolonged by 1 month. Yet the efficiency was measured to be 3.2 KLOC/PM.

However, the effectiveness of the inspections was only 0.4 instead of the planned 0.5. Therefore, further analysis was conducted. It turned out that the experience of the designers was not considered in the model of effectiveness. In all projects conducted in the "ABS" group, the designers had a medium level of experience, whereas the designers in the "Maui" project had only little experience.

10.5.2 Package

The project characterization which is the result of the post-mortem analysis, the gathered quantitative experiences, and the tailored process model "Standard 4711 with design insp.+review" become new cases. The relationships to existing cases are also specified (Figure 14; the relationships are indicated by connectors to figures 11 and 13).

Since a new important applicability factor (design experience) was identified, all existing project characterizations are extended by this new attribute (see gray texts in figures 11, 13, and 14). For "Maui" the attribute value is "low", whereas for "Hugo" and "Judy" as well as their generalization the attribute value is "medium". For all other projects, the attribute value is set to "unknown", because it would require too much effort to collect this information. Moreover, this information would be impossible to get (at least in part), since some of the old project managers have already left the software development organization.

10.6 Strategic Aftermath

Looking at the inspection effectiveness and the project efficiency, no conclusive evaluation can be done with respect to the hypothesis that inspections increase project efficiency, because "Maui" could be an outlier regarding efficiency. However, the 3.2 KLOC/PM are quite promising. Therefore, further empirical evidence is needed. For this reason, the "fuel injection" group creates a new process model "Standard 4711 with design insp.". This process model shall be applied in the next three "fuel injection" projects in order to be able to build a more valid efficiency model.

It is also expected that the inspection effectiveness will be better if more experienced designers are part of the project. Therefore, inspection effectiveness will also be measured in future projects.

10.7 Conclusion

The scenario illustrates that:

- An experience base can supply knowledge users did not expect (in the scenario, the project manager did not know that design reviews were performed only in the “ABS” group).
- Goal-oriented, organizational learning leads to strategically relevant knowledge faster than learning on the level of individuals or groups (see strategic aftermath). Even if the explicitly available knowledge does not meet the current needs perfectly, it can be taken advantage of (see utilization of “design review” experience in the “ABS” group).
- Software engineering knowledge is not static. It is complemented on a continuous basis. In the scenario, both concrete cases and structural knowledge (e.g., “design experience” is relevant for selecting the right effectiveness model) is added. This mirrors the experimental approach Fraunhofer IESE takes.