



IST PROJECT 35111

Tightening knowledge sharing in distributed software communities by applying semantic technologies

Project Number:	35111
Project Acronym:	TEAM
Project Title:	Tightening knowledge sharing in distributed software communities by applying semantic technologies
Instrument:	STREP
Thematic Priority:	Information Society Technologies (IST)
Start date of the project:	September 1 st , 2006
Duration:	30 months

D1: Report describing state-of-the art KM in Software Engineering

Lead contractor:	ICCS
Editor(s):	Spyros Ntioudis ,Walid Maalej, Hans-Joerg Happel
Author(s):	Spyros Ntioudis, Walid Maalej, Hans-Joerg Happel, Dimitris Panagiotou
Submission date:	December 2006
Dissemination level:	Public

Abstract: This report describes the state of the research and practice in the areas of Knowledge Management in Software Engineering. Special emphasis is laid upon specific knowledge representation and reasoning requirements coming from the open-source communities and outsourced software development.

Versioning and Contribution History

Version	Date	Modification reason	Modified by
0.1	2/11/06	Creation of Sections 2 (Introduction), 3.1 (KM approaches in SE) and 6 (Challenges and future work)	Spyros Ntioudis
0.1	2/11/06	Creation of Section 3.2 (Tools for KM in SE)	Dimitris Panagiotou
0.2	3/11/06	Creation of Section 4 (Knowledge representation)	Walid Maalej
0.3	8/11/06	Creation of Section 5 (Empirical results)	Hans Joerg Happel
0.4	27/11/06	Comments on the overall report	Maria Legal
0.5	5/12/06	Comments on the overall report	Roman Schmidt
0.6	6/12/06	Comments integration and finalisation of Section 5	Hans Joerg Happel
0.7	8/12/06	Comments integration and finalisation of Section 4	Walid Maalej
0.8	9/12/06	Comments integration and finalisation of Sections 2, 3, 6	Spyros Ntioudis, Dimitris Panagiotou
0.9	10/12/06	Creation of Section 1 (Executive Summary)	Spyros Ntioudis
0.95	11/12/06	Proof reading and formatting issues	Spyros Ntioudis
0.98	13/12/06	Proof reading & final comments	Maria Legal
1.0	15/12/06	Final version	Spyros Ntioudis

This document has been produced in the context of the TEAM Project. The TEAM project is part of the European Community's Sixth Framework Program for research and development and is as such funded by the European Commission. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors view.

Table of Contents

1	EXECUTIVE SUMMARY	7
2	INTRODUCTION.....	10
2.1	DEFINITIONS	10
2.2	MOTIVATION FOR KM IN SE.....	11
3	KM APPROACHES & TOOLS IN SE.....	14
3.1	KM APPROACHES IN SE	14
3.1.1	<i>EXPERIENCE FACTORY</i>	<i>14</i>
3.1.2	<i>KNOWLEDGE DUST TO PEARLS APPROACH</i>	<i>15</i>
3.1.3	<i>PERSONAL SOFTWARE PROCESS</i>	<i>16</i>
3.1.4	<i>TEAM SOFTWARE PROCESS.....</i>	<i>17</i>
3.1.5	<i>SOFTWARE ENGINEERING CONSORTIA.....</i>	<i>19</i>
3.1.6	<i>INITIATIVES FOR STANDARDS.....</i>	<i>20</i>
3.1.7	<i>PROCESS-BASED KNOWLEDGE MANAGEMENT SUPPORT FOR SOFTWARE ENGINEERING.....</i>	<i>22</i>
3.2	KM TOOLS IN SE.....	22
3.2.1	<i>DOCUMENT MANAGEMENT TOOLS</i>	<i>22</i>
3.2.2	<i>COMPETENCE MANAGEMENT TOOLS.....</i>	<i>23</i>
3.2.3	<i>INTELLIGENT REQUIREMENTS ASSISTANTS</i>	<i>24</i>
3.2.4	<i>KNOWLEDGE BASED PROGRAM DESIGNERS.....</i>	<i>24</i>
3.2.5	<i>KNOWLEDGE BASED CODE GENERATORS AND RECOMMENDATION SYSTEMS.....</i>	<i>25</i>
3.2.6	<i>SMART CODE ANALYSIS TOOLS</i>	<i>26</i>
3.2.7	<i>SOFTWARE MAINTENANCE TOOLS.....</i>	<i>26</i>
3.2.8	<i>WIKIS</i>	<i>27</i>
3.2.9	<i>COLLABORATIVE TOOLS</i>	<i>28</i>
4	KNOWLEDGE REPRESENTATION	29
4.1	REPRESENTING KNOWLEDGE ABOUT SYSTEM MODELS.....	29
4.1.1	<i>MODELS AND NOTATIONS.....</i>	<i>29</i>
4.1.2	<i>ONTOLOGY-BASED SOFTWARE ENGINEERING</i>	<i>32</i>
4.2	REPRESENTING DEVELOPMENT KNOWLEDGE	33
4.2.1	<i>RATIONALE MANAGEMENT</i>	<i>33</i>
4.2.2	<i>CAPTURING RATIONALE</i>	<i>34</i>
5	EMPIRICAL RESULTS	37
5.1	PERSONAL SPHERE	37
5.2	TEAM SPHERE	39
5.2.1	<i>EXPLORATORY STUDIES</i>	<i>39</i>
5.2.2	<i>SPECIFIC INVESTIGATIONS.....</i>	<i>40</i>
5.2.3	<i>SUMMARY.....</i>	<i>40</i>
5.3	DISTRIBUTED DEVELOPMENT	41
5.3.1	<i>EXPLORATORY STUDIES</i>	<i>42</i>
5.3.2	<i>SPECIFIC INVESTIGATIONS.....</i>	<i>42</i>

5.3.3	<i>SUMMARY</i>	43
5.4	OPEN SOURCE DEVELOPMENT.....	44
5.5	SUMMARY.....	45
6	CHALLENGES & FUTURE WORK	46
7	REFERENCES	48

List of Figures

Figure 1: The Quality Improvement Paradigm (from [10]).....	14
Figure 2: The Experience Factory Model	15
Figure 3: The Experience Factory Model with the new feedback loop of the Knowledge Dust to Pearls approach.	16
Figure 4: The TSP process	18
Figure 5: Taxonomy of UML diagrams represented as a UML class diagram	31
Figure 6: Rationale objects	34
Figure 7: Rationale use in Software Engineering (from [78])	36
Figure 8: Relationships between developers in a team	39
Figure 9: Relationships between developers in a distributed team.....	41
Figure 10: Relationships between developers in an Open Source team	44

List of Tables

Table 1: The SWEBOK Knowledge Areas	21
Table 2: Summary of typical competence management tools' features	24
Table 3: Research in individual developers' work practices	38
Table 4: Research in team software development	41
Table 5: Research in distributed/global software development.....	44
Table 6: Research in Open Source work practices	45
Table 7: Predominant/specific knowledge issues in software development spheres.....	46

1 EXECUTIVE SUMMARY

Knowledge Management (KM) refers to a range of practices used by organisations to identify, create, represent, and distribute knowledge for reuse, awareness and learning across the organisation. On the other hand, Software Engineering (SE) is the design, development, and documentation of software by applying technologies and practices from computer science, project management, engineering and other fields.

The nature of Software Engineering is rather complex since it involves many people working in different phases and activities. It is a field where constant technology changes take place rendering the work of the involved people extremely dynamic, in the sense of discovery and solution of new problems on a daily basis. The knowledge in Software Engineering is diverse and its proportions immense and continuously expanding. Thus, a structured way of managing the knowledge and treating the knowledge and its owners as valuable assets could help organisations leverage the knowledge they possess. The most important needs that drive the use of KM in SE are:

- Capturing and sharing process and product knowledge.
- Acquiring knowledge about the application domain.
- Acquiring knowledge about new technologies.
- Knowing who knows what.
- Distance collaboration.

There exist several methodical approaches that deal with managing knowledge in Software Engineering environments (see Section 3.1). The most important ones consist of:

- The Experience Factory [11].
- The Knowledge Dust to Pearls approach [13].
- The Personal Software Process [14].
- The Team Software Process [15].
- The formation of Software Engineering Consortia such as the Software Experience Center (SEC) Consortium and the Consortium for Software Engineering Research (CSER).
- Initiatives for standards such as the Software Engineering Body of Knowledge (SWEBOK) and the Center for Empirically-Based Software Engineering (CeBASE).
- Process-based Knowledge Management support for Software Engineering [18].

In addition, a number of representative KM tools for assisting Software Engineering tasks is given in Section 3.2, categorised according to the following:

- Document management tools
- Competence management tools
- Intelligent requirements assistants
- Knowledge-based program designers

- Knowledge-based code generators and recommendation systems
- Smart code analysis tools
- Software maintenance tools
- Wikis
- Collaborative tools

Knowledge representation is a key issue in successfully applying any KM program in an organisation. For representing knowledge about software systems, we demonstrate in Section 4 how well known modelling approaches as well as the emerging ontology-based software development are both complementary and most adequate to the nature of software. Modelling is a perfect toolkit for externalising, formalising and communicating knowledge about complex and manifold software systems, even if the SE communities do not explicitly consider it as a knowledge representation method. Moreover, researches showed similarities between models and ontologies and how they can be brought together to tap the full potential of smart Software Engineering processes and environments. First researches in ontology-based Software Engineering are promising. However, this field is still in an early experimentation phase but is expected to become one of the most agile in the near future.

In addition, another facet of knowledge representation is rationale and the way rationale has to be represented. Rationale management addresses development and collaboration knowledge mostly resulting from debates and argumentation. Investigating the most relevant rationale representation approaches (Issue-Based Information System, Question, Option and Criteria, Decision Representation Language and NFR Framework) we show the clear similarities between all these methods. Recent researches in this field extended the focus of rationale from the design phase to all software life cycle activities and show how rationale can be useful for all decisions during these activities.

Empirical works related to knowledge sharing are presented in Section 5. They are analysed according to the spheres of the individual developer, team development, distributed development and Open Source development. While each of those spheres introduces new knowledge sharing needs (e.g. task coordination) the channels for knowledge exchange get more restricted. The empirical results from the investigated studies show that (in general) developers strongly prefer informal face-to-face communication for knowledge sharing. Nevertheless, practices from Open Source development show that tools and artefacts can partially compensate a lack of informal and formal communication. Notably, most of the empirical works focus on low level indicators for information exchange in development teams like communication patterns. There is a lack of theoretically grounded conceptualisations of knowledge sharing in distributed teams. This is particularly important since studies show that knowledge sharing needs differ in terms of project stage, developer experience and tool support.

Finally, in Section 6, the challenges that Knowledge Management has to face upon its implementation in software organisations are presented. The most significant ones include:

- The lack of time in searching for and creating new knowledge.
- The elusive nature of software itself resulting in being less reused than expected.
- The possessive attitude of the developers with respect to their knowledge and their reluctance in sharing it by fear that they will become expendable.

TEAM intends to address the aforementioned challenges and provide:

- An automatic elicitation of the knowledge through the concept of a context observer that is integrated in a software development environment (IDE).
- A distributed organisation of the knowledge (by using P2P infrastructures).
- An efficient search mechanism that enables finding similar but for the given context very relevant knowledge items.
- Proactive and personalised knowledge delivery depending on a user's working and personal context (provided in an unobtrusive manner).

2 INTRODUCTION

This report describes the state of the research and practice in the areas of Knowledge Management (KM) in Software Engineering (SE). Special emphasis is laid upon specific knowledge representation and reasoning requirements coming from the open-source communities and outsourced software development.

The structure of this report is as follows: The present section constitutes the introduction, providing the definitions of the two areas under consideration, namely of Knowledge Management and of Software Engineering. It also delves in the motivation for applying Knowledge Management techniques in order to enhance some (or all) of the phases of Software Engineering. In Section 3 the KM approaches for Software Engineering and the respective tools are presented, albeit not exhaustively as far as the latter are concerned but citing the most representative ones. Section 4 comprises the most common techniques of representing knowledge in Software Engineering and Section 5 gives an overview about empirical works related to Knowledge Management practices in Software Engineering. Finally, Section 6 presents the general challenges regarding KM in SE as well as the ones that the consortium will have to address in the context of the TEAM project.

2.1 Definitions

*“Knowledge Management refers to a range of practices used by organisations to identify, create, represent, and distribute knowledge for reuse, awareness and learning across the organisation”*¹. Knowledge transfer (one aspect of Knowledge Management) has always existed in one form or another, for example through on-the-job peer discussions, formal apprenticeship, corporate libraries, professional training, and mentoring programmes. However, since the late twentieth century additional technology has been applied to this task, such as knowledge bases, expert systems, and knowledge repositories. Knowledge Management initiatives (programs) attempt to manage the process of creation or identification, accumulation, and application of knowledge or intellectual capital (i.e. the intangible assets of a company which contribute to its valuation) across an organisation.

*“Software engineering is (1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software, and (2) the study of approaches as in (1)”*². A less formal definition is referring to Software Engineering as *“the design, development, and documentation of software by applying technologies and practices from computer science, project management, engineering, application domains, interface design, digital asset management and other fields”*³.

¹ http://en.wikipedia.org/wiki/Knowledge_management

² Definition by IEEE (<http://www.ieee.org/>)

³ http://en.wikipedia.org/wiki/Software_engineering

Knowledge management in the context of software development organisations is described by Davenport and Prusak [1] as *“a fluid mix of framed experience, values, contextual information, and expert insights and grounded intuitions that provides a framework for evaluating and incorporating new experiences and information. It originates and is applied in the minds of the knower. In software organisations, knowledge often becomes embedded not only in documents or repositories, but also in organisational routines, processes, practices, and norms”*.

The first argument in favour of managing knowledge in Software Engineering is that Software Engineering is a human and knowledge intensive activity [2]. Software development is also a process that involves a great deal of decision making with several possible alternatives to follow. Thus, it is simply insufficient to rely solely on personal knowledge and experience to tackle decision making in Software Engineering. It is considered more a group activity where individuals need to communicate and coordinate, sharing individual knowledge and leveraging it at a project and organisation level. This is exactly what Knowledge Management proposes shifting the focus to collective creativity, exploiting the emerging behavioural idea – “none of us is as smart as all of us” [3].

The following section presents the needs related to Knowledge Management in the Software Engineering domain, namely the motivation for KM in SE.

2.2 Motivation for KM in SE

The nature of Software Engineering is rather complex since it involves many people working in different phases and activities. It is a field where constant technology changes take place rendering the work of the involved people extremely dynamic, in the sense of discovery and solution of new problems on a daily basis. The knowledge in Software Engineering is diverse and its proportions immense and continuously expanding. Organisations face the problems of keeping track of what this knowledge is, where it is, and who has it. Thus, a structured way of managing the knowledge and treating the knowledge and its owners as valuable assets could help organisations leverage the knowledge they possess. The most important needs that drive the use of KM in SE are the following:

- Capturing and sharing process and product knowledge.

Software product and process differ from each other in terms of goals and contexts. It is simply unrealistic to assume that the same software development approach is applicable for all projects or products [4]. According to [5] this is one of the reasons that make the software discipline inherently experimental and, thus, experience is constantly enhanced with each development project. *“Knowledge emerges in work practices, often being defined by the first project to address the issues involved”* [8]. The ideal way to go forward would be to apply past, accumulated experience to future projects in order to avoid mistakes and leverage successes and that software development teams benefit from existing experience. Unfortunately, this is not always the case since, often, past work practices are not captured [8] resulting in development teams reinventing the wheel and repeating the same mistakes that have been resolved in the past [6, 7]. The aforementioned problems are also intertwined with the problem of transferring knowledge to new members in the organisation. Knowledge Management is ideal in comprising the means to address the issues of capturing and sharing knowledge in the Software Engineering domain.

- Acquiring knowledge about the application domain.

It is indispensable that software development teams possess adequate knowledge with respect to the domain for which software is being developed. It is rather frequent that a new domain requires learning a specific technique or a new programming language or application of a new kind of project management technique, resulting in longer times with respect to acquisition of the required experience and skills [7]. The ways in which domain knowledge, that no one in the organisation possesses, can be acquired is either by training or by hiring knowledgeable employees. Knowledge Management can, however, play an important role in assisting the organisation of knowledge acquisition and the identification of expertise as well as capturing, packaging and sharing knowledge that already exists in the organisation.

- Acquiring knowledge about new technologies

The pace of change in Software Engineering technologies is fast resulting in difficulties in keeping up with the latest changes. On one hand, the emergence of new technologies makes software more powerful, but on the other every emerging technology cannot be mastered overnight. It is evident that accurately estimating the cost of a project is a very difficult or even, sometimes, not feasible task when the technologies to be employed are new, not adequately tested and may even change during the project's lifecycle. The "learning by doing" approach is where software engineers resort to when dealing with new technologies for which little or no knowledge exists in the heads of the development team(s) [7]. This often results in serious delays of projects. Knowledge Management's role in this specific case is to foster a knowledge sharing culture within the organisation in order to help facilitate sharing of knowledge related to new technologies. Knowledge sharing can be promoted via communities of practice and interests, thus speeding up the learning curve.

- Knowing who knows what

"It is more important for Software Engineering organisations to exploit and manage their intangible assets in contrast to their physical assets" [9]. The intangible assets comprise in this case the tacit knowledge that resides in the heads of the members of a Software Engineering organisation. Therefore, it is imperative that the organisation invests in management of those intangible assets, i.e. possessing knowledge about who knows what. This is part of competence management and is the right direction towards reducing the time required to seek and retrieve experts in a specific field. Experts can help not only in resolving a potential problem but also (or instead) in pointing to the right piece of information that is pertinent to the respective problem. The issue of time invested in seeking the right information is of exceptional importance for the unhindered execution of a project. This is corroborated by the results of a study reporting that people in software organisations spent 40% of their time in searching for and accessing different types of information related to their projects [8]. *"Software organisations are heavily dependent on tacit knowledge, which is very mobile"* [9]. In order to avoid or, at least, mitigate the risk of creation of severe knowledge gaps in cases of people suddenly leaving the organisation, it is imperative to know what knowledge they possess [6]. Thus, retaining, if possible, the people who possess critical knowledge about processes and practices, can be of utmost importance, even for the viability of the organisation. Knowledge Management comes into play by assisting in building structures and frameworks for

capturing key information that can help retain some knowledge when employees leave.

- Distance collaboration

As far as large scale software development is concerned, this is inherently a group activity. The division of work into phases, almost always, presupposes the assignment of different phases to different groups and their involvement either at the same point in time or at a later one. These groups can also be based in different geographic locations and it is common that group members live and work in different time zones. Outsourcing of subsystems to subcontractors also results in geographically co-located teams that need to communicate, collaborate, and coordinate independently of time and place. Knowledge Management acknowledges the need to capture, organise and store knowledge, as well as the necessity of knowledge transfer. Therefore:

- Communication in Software Engineering is often related to the transfer of knowledge.
- Collaboration is related to mutual sharing of knowledge.
- Coordination that is independent of time and space is facilitated if the work artefacts and their status are stored and made part of a knowledge repository.

3 KM APPROACHES & TOOLS IN SE

3.1 KM Approaches in SE

3.1.1 Experience Factory

As far as software development is concerned, the most common way of learning is during projects. Consequently, the need for capturing that kind of knowledge arises in order to assist organisational learning. The knowledge from all projects ought to be documented, collected and organised into a structured repository aiming at supporting posterior decision making with respect to future projects.

A concept that supports this idea is the Quality Improvement Paradigm (QIP) [10]. Figure 1 illustrates the way learning occurs at each project level by analyzing and drawing conclusions about the project's results, both during execution and post mortem. These results are then analysed for a second time by means of abstracting from the specific problem domain, subsequently packaged in an organisational context and stored in an experience database. The resulting experience repository will support the planning process regarding future projects (i.e. selecting the processes, methods, techniques and tools that have been proved to be useful to the organisation in past projects).

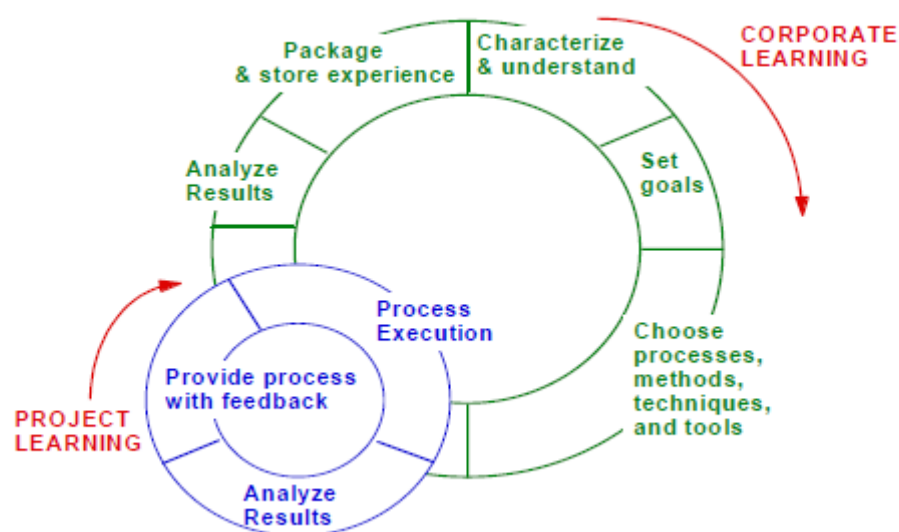


Figure 1: The Quality Improvement Paradigm (from [10])

The Experience Factory [11] is a framework for the implementation of the Quality Improvement Paradigm. The approach has been successfully applied to software development at NASA for more than 25 years and recently at other organisations. The Experience Factory enables organisational learning and professes the need of existence of a separate support organisation. The role of the support organisation is to support the project organisation in order to manage and learn from its own experience. This goal is achieved by assisting the project organisation in observing and collecting data about itself, building models and drawing conclusions based on that data. The collected experience is

subsequently stored in packages for further reuse, and most importantly, it is fed back to the project organisation (see Figure 2).

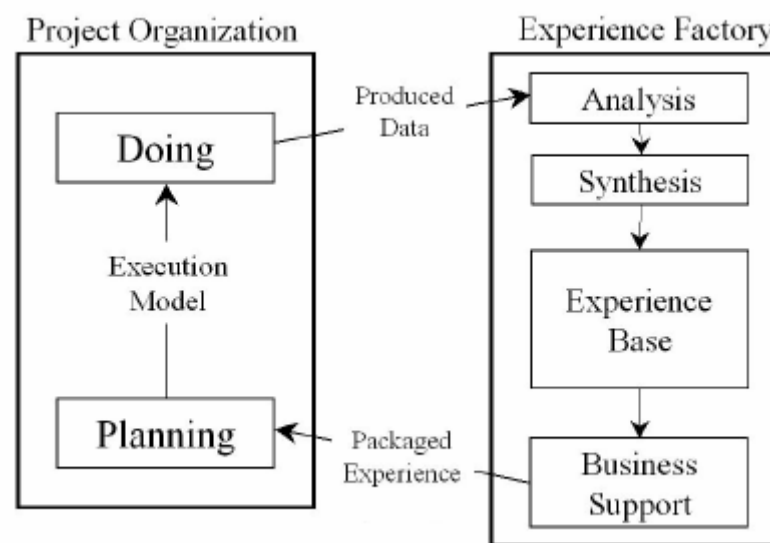


Figure 2: The Experience Factory Model

The Experience Factory approach takes into account the software discipline's experimental, evolutionary, and non repetitive characteristics. It has components that address capturing, storing, distributing, applying, and creating new experience. It also has components that address analysis and synthesis of knowledge [6]. Organisations for which software development is not their core business process but whose processes are design-oriented rather than production-oriented (e.g. manufacturing), have recently been found capable of implementing successfully custom versions of the Experience Factory approach, thus having as a primary benefit their transformation into learning organisations. In summary, the Experience Factory is an example of an approach that is based upon the assumption that knowledge and experience can be made explicit so that they can be stored in knowledge and experience bases.

3.1.2 Knowledge Dust to Pearls approach

The Knowledge Dust to Pearls approach is influenced by the ideas of the Quality Improvement Paradigm (QIP) (see Section 3.1.1) which provides a model for process improvement in software organisations. QIP uses the notions of continuous improvement and iterations as the main vehicle for planning, executing, evaluating, and improving processes. The backbone of the Knowledge Dust to Pearls approach is the Experience Factory (see Section 3.1.1), which establishes a learning organisation. The Experience Factory is, however, a sophisticated approach that satisfies an organisation's long-term needs of sharing experience. A complementary approach that satisfies the short-term needs of an organisation is the AnswerGarden approach [12].

The AnswerGarden approach lets employees store and organise questions and answers as they are received and answered by the organisation. By storing questions and corresponding answers in a common repository, the knowledge can easily be spread throughout the organisation.

The Knowledge Dust to Pearls approach [13] combines and makes use of benefits both from the AnswerGarden (which represents Knowledge Dust) and the Experience Factory (which represents Knowledge Pearls). It captures the knowledge dust that employees use and exchange on a daily basis and immediately, with minimal modifications, makes it available throughout the organisation. This process is accomplished by creating a system that supports peer-to-peer activities; i.e. the employees of the organisation help each other and fulfil the short-term return goals of a knowledge capturing and sharing approach.

The Knowledge Dust to Pearls approach adds a new and shorter feedback loop to the Experience Factory, which can be seen in Figure 3.

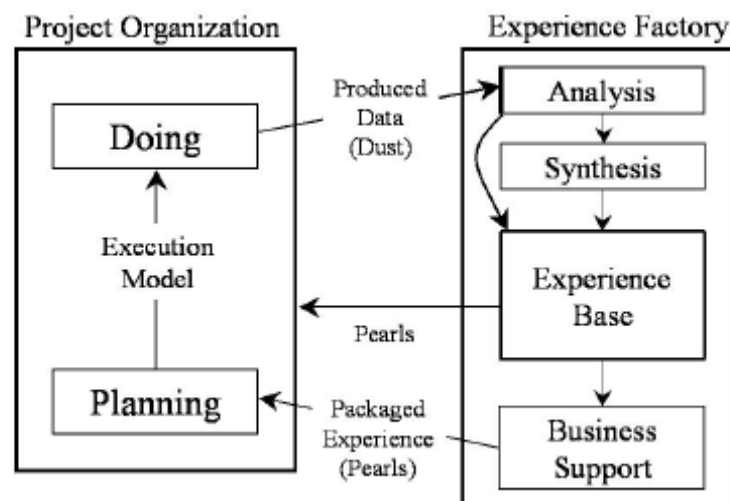


Figure 3: The Experience Factory Model with the new feedback loop of the Knowledge Dust to Pearls approach.

What the above figure illustrates is that in the Knowledge Dust to Pearls approach the data about an organisation, i.e. the dust, goes through a minimal analysis phase turning it into a mini-pearl. The mini-pearl entirely bypasses the synthesis phase and is stored in the experience base, thus it is made available to the project organisation almost immediately after collection. In this way, the organisation receives benefits from the very beginning, as soon as the dust collection process is established.

3.1.3 Personal Software Process⁴

It is a reality that most software-development groups are significantly late and over budget regarding more than half of all software projects they undertake, and nearly a quarter of those projects are cancelled without ever being completed. Although developers recognise that unrealistic schedules, inadequate resources, and unstable requirements constitute often the main causes for such failures, only a minority of them knows how to solve these types of problems. Watts S. Humphrey and the Software Engineering Institute (SEI) profess that the Personal Software Process (PSP) is a clear and proven solution to the aforementioned

⁴ <http://www.sei.cmu.edu/tsp/psp.html>

problems. PSP [14] is a self-improvement process for software engineers comprising precise methods developed over many years by SEI. The PSP has successfully transformed work practices in a wide range of organisations and has already produced some promising results⁵.

PSP training focuses on the skills required by individual software engineers to improve their personal performance. Once learned and effectively applied, PSP-trained engineers are qualified to participate on a team using the Team Software Process (TSP) (see Section 3.1.4).

PSP presents a disciplined process for software engineers and anyone else involved in software development. This process includes defect management, comprehensive planning, and precise project tracking and reporting. The goal for PSP is to provide to developers exactly what they need in order to deliver quality products on predictable schedules.

The Personal Software Process can be applied to many parts of the software development process, including:

- small-program development
- requirement definition
- document writing
- systems tests
- systems maintenance
- enhancement of large software systems

3.1.4 Team Software Process⁶

The Team Software Process (TSP) [15], along with the Personal Software Process, helps the high-performance engineer to

- ensure quality software products,
- create secure software products,
- improve process management in an organisation.

Engineering groups use the TSP to apply integrated team concepts to the development of software-intensive systems. The TSP provides team projects with explicit guidance on how to accomplish their objectives. As shown in Figure 6, the TSP guides teams through the four typical phases of a project. These projects may start or end on any phase, or they can run from beginning to end. Before each phase, the team goes through a complete launch or relaunch, where they plan and organise their work. Generally, once team members are PSP trained, a four-day launch workshop provides enough guidance for the team to complete a full project phase. Teams then need a two-day relaunch workshop to kick off the second and each subsequent phase. These launches are not training; they are part of the project.

⁵ For more information: <http://www.sei.cmu.edu/tsp/results/results.html>

⁶ <http://www.sei.cmu.edu/tsp/tsp.html>

To start a TSP project, the launch process script leads teams through the following steps [16]:

- Review project objectives with management.
- Establish team roles.
- Agree on and document the team's goals.
- Produce an overall development strategy.
- Define the team's development process.
- Plan for the needed support facilities.
- Make a development plan for the entire project.
- Make a quality plan and set quality targets.
- Make detailed plans for each engineer for the next phase.
- Merge the individual plans into a team plan.
- Rebalance team workload to achieve a minimum overall schedule.
- Assess project risks and assign tracking responsibility for each key risk.
- Hold a launch post mortem.

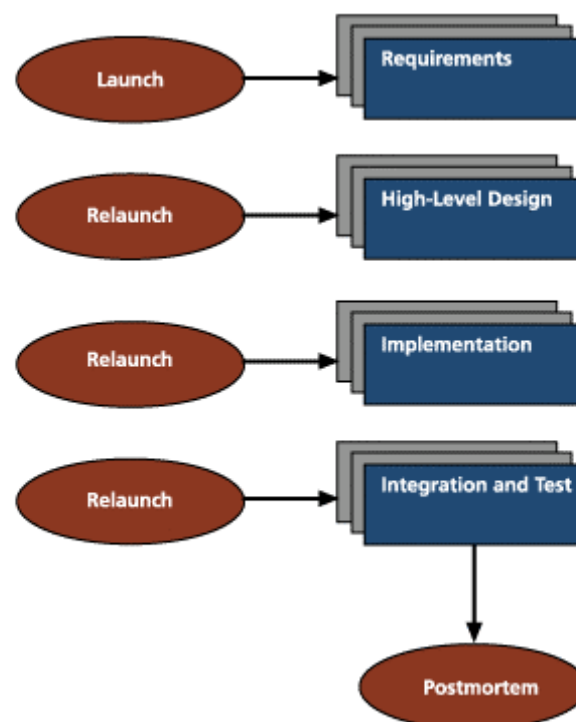


Figure 4: The TSP process

In the final launch step, the team reviews its plans and the project's key risks with management. Once the project starts, the team conducts weekly team meetings and periodically reports its status to management and to the customer.

After the launch, the TSP provides a defined process framework for managing, tracking and reporting the team's progress. Using TSP, an organisation can build self-directed teams that plan and track their work, establish goals, and own their processes and plans. TSP can provide assistance to organisations towards establishing a mature and disciplined engineering practice that produces secure, reliable software.

3.1.5 *Software Engineering Consortia*

It is a fact that software organisations learn not only from their own experiences, but also from external sources (e.g. software vendors and other software development companies). Organisations whose core business activity is software development or who have a significant software component embedded in their products (such as avionics, telecommunication, and automobile industries) have formed Software Engineering consortia aiming towards facilitating inter-company learning. Software engineering consortia is a systematic and well defined form of communication and knowledge sharing. Examples of such communities are:

- The Maryland Software Industry Consortium (SwIC)⁷ Project, in conjunction with the Maryland Department of Business and Economic Development, provides a Software Engineering resource to assist Maryland organisations in advancing the practices of system and Software Engineering and improving the quality of their software related products and services. SwIC integrates research and experience into practical improvement, creates opportunities to develop and disseminate improvement practices, enhances the competitiveness of member companies, especially small to medium-sized companies, accelerates new software technology adaptation, leverages member company experience, promotes inter-corporate cooperation of member organisations, and provides training and education.
- The Software Experience Center (SEC) Consortium⁸ is a joint project between the Fraunhofer Center–Maryland⁹ and the Fraunhofer IESE¹⁰ and its goal is to improve the software competencies and development practices of member companies. To achieve this goal, member companies share past and ongoing experiences in software process improvement and particular development technologies. The Fraunhofer organisations contribute their expertise to help analyze, package, and disseminate the lessons to be learned from these experiences. They collaborate to provide a number of services to member companies, namely they organise workshops to provide a forum for the discussion of software development experience, produce a series of experience reports that address specific technologies of interest to the Consortium and have developed an extensive network of software experts, both within the organisations and externally, that can be made available to SEC member companies. The Consortium is currently composed of five international corporations with

⁷ <http://fc-md.umd.edu/fcmd/Apps/ProjectsRecord.asp?ID=12>

⁸ <http://fc-md.umd.edu/fcmd/Apps/ProjectsRecord.asp?ID=14>

⁹ <http://fc-md.umd.edu/fcmd/index.html>

¹⁰ <http://www.iese.fraunhofer.de/fhg/iese/>

significant investments in software development: ABB, Boeing, DaimlerChrysler, Motorola, and Nokia.

- The Software Program Managers Network (SPMN)¹¹, whose purpose is to “*seek out proven industry and government software best practices and convey them to managers of large-scale United States Department of Defense software-intensive acquisition programs.*”
- The Consortium for Software Engineering Research (CSER)¹² in Canada was established in 1996 and, in the years since its inception, has been recognised for its significant contributions to the Canadian software industry and to Software Engineering knowledge. CSER has also played a major role in the creation and improvement of Software Engineering programs in Canadian Universities. CSER’s two goals are to undertake collaborative research with university investigators into critical industrial problems and to raise the profile of Software Engineering technology in the university curriculum. The CSER research program will improve and expand the methodologies, tools and techniques used to construct, deploy, support and evolve software; and to improve the quality of software, increase productivity, and manage costs

3.1.6 Initiatives for standards

Industry-wide experience and knowledge leverage of all software development organisations is pursued and achieved also through committees or groups of experts that identify patterns (e.g. software design patterns [17]) and generate handbooks and standards pertinent and applicable to software development (e.g. IEEE and ISO standards).

The Software Engineering Body of Knowledge (SWEBOK) is one such initiative and a product of the Software Engineering Coordinating Committee¹³. The IEEE Computer Society¹⁴ is also involved. The body of knowledge represents what a practicing software engineer needs to master on a daily basis. SWEBOK is intended to serve as a guide to the knowledge already existing in the form of the accumulated Software Engineering literature and aims at documenting fundamental knowledge that is relatively stable over time instead of knowledge that is subject to rapid technological changes.

The Guide to the Software Engineering Body of Knowledge (SWEBOK) was established with the following five objectives:

1. To promote a consistent view of Software Engineering worldwide.
2. To clarify the place - and set the boundary - of Software Engineering with respect to other disciplines such as computer science, project management, computer engineering, and mathematics.

¹¹ <http://www.spmn.com/>

¹² <http://www.cser.ca/>

¹³ <http://www.acm.org/serving/se/>

¹⁴ <http://www.computer.org/>

3. To characterise the contents of the Software Engineering.
4. To provide a topical access to the Software Engineering Body of Knowledge.
5. To provide a foundation for curriculum development and for individual certification and licensing material discipline.

The material that is recognised as being within the Software Engineering discipline is organised/subdivided into ten Knowledge Areas listed in Table 1.

Software requirements
Software design
Software construction
Software testing
Software maintenance
Software configuration management
Software engineering management
Software engineering process
Software engineering tools and methods
Software quality

Table 1: The SWEBOK Knowledge Areas

Another kind of initiatives and projects aims at building knowledge bases for empirical Software Engineering. The need for such kinds of projects and initiatives stems from the fact that software development is a people- and knowledge- intensive activity. As such, it is a rapidly changing field, and although it is slowly maturing, many activities are still ad hoc and depend upon personal experiences. Consequently, software-developing organisations need assistance in setting up and running increasingly critical projects and, in order to reach their goals, it is imperative that software development teams understand and select the right models and techniques to support their projects.

The Center for Empirically-Based Software Engineering (CeBASE)¹⁵ was organised to support software organisations in dealing efficiently with the aforementioned problem areas. CeBASE accumulates empirical models in order to provide validated guidelines for selecting techniques and models, recommend areas for research, and support Software Engineering education. CeBASE's objective is to transform Software Engineering *“to an engineering-based discipline in which development processes are selected based on what is known about their effects on products, through synthesis, derivation, organisation, and dissemination of empirical knowledge on software development and evolution phenomenology”*.

¹⁵ [http:// www.cebase.org/](http://www.cebase.org/)

3.1.7 Process-based Knowledge Management support for Software Engineering

The primary goal of Process-Oriented Knowledge Management (POKM) is to establish, run and maintain an organisational environment that provides process participants with the information needed to successfully perform their activities as defined in the process model.

Holz presents in [18] a detailed life-cycle model for POKM that is specific to software development processes. This life-cycle model for Software Engineering Process-Oriented KM (SE-POKM) is integrated into the life-cycle model performed by the organisation's Process Group and becomes an essential part of a continuous organisational learning process.

The SE-POKM model encompasses the following:

- An explicit representation of dynamic (i.e. situation-specific) information needs that typically arise for process participants during software development activities; this representation also covers potential ways to satisfy those information needs.
- A specification of the information need retrieval during process enactment. Depending on a characterization of their current situation (i.e. current activities, individual preferences and skills etc.), process participants are provided with modelled information needs that are expected to arise for them during their activities; in particular, corresponding selected information items are retrieved for each of these information needs, which are assumed to satisfy the information needs in required detail.
- A guideline for the experience packaging phase based on feedback from process participants; during this phase, the initial model of relevant and useful information needs is updated to better reflect the participants' actual information needs.

In order to automate the retrieval of information needs and corresponding information items, Holz also presents the Process-oriented Information resource Management Environment (PRIME). PRIME provides a technical infrastructure for knowledge distribution and feedback communication and is designed to be coupled with the organisation's Process-Centred Software Engineering Environment.

3.2 KM Tools in SE

Software engineering involves a great variety of knowledge-intensive tasks: e.g. analysing user requirements for new software systems, identifying and applying best software development practices, collecting experience about project planning and risk management, etc. [2]. In this section, an overview of representative KM tools for assisting Software Engineering tasks is given categorised by a classification adapted by Rus et al. ([19]).

3.2.1 Document Management Tools

The results of all Software Engineering tasks are documents (even source code and executable programs can be regarded as documents). Because Software Engineering is so dominated by the documents, Rus et al. in [19] argue that the foundation for a Knowledge Management system in Software Engineering is a document management system (DMS).

According to wikipedia [21], a document management system is a computer system (or set of computer programs) used to track and store electronic documents and/or images of paper documents. Document management systems commonly provide storage, versioning,

metadata, security, as well as indexing and retrieval capabilities. The term has some overlap with the concepts of Content Management Systems and is often viewed as a component of Enterprise Content Management Systems and related to Digital Asset Management. A list of components provided by a typical DMS should include most of the following: metadata, capture, indexing, storage, retrieval, distribution, security, workflow and versioning.

A very popular and powerful DMS is “Hyperwave IS/6”. According to NewHyperG AG [22] it provides, among others, document management with version control, support for multiple file formats, archiving with configuration management, search in arbitrary indexed meta data, user and group management with LDAP and Active Directory, finding similar documents and search in external sources (e.g.: of Web spiders, databases, file server).

Although Document Management systems support Software Engineering tasks, it is interesting to investigate how these technologies apply to Software Engineering documents. The Software Concordance for example, is a prototype integrated development environment (IDE) that uses a tree-based document representation for software documents, an integration between hypermedia and program analysis services, and inline multimedia documentation in program source code to improve software document management [23]. Another example is srcML (SouRCe Markup Language). srcML presumes a document view of source code where information about the syntactic structure is layered over the original source code document. The resultant multi-layered document has a base layer of all the original text (and formatting). The second layer is the syntactic information, derived from the grammar of the programming language, and is encoded in XML, thus enabling tasks such as analysis and transformation of the source code [24].

3.2.2 Competence Management Tools

A definition for competence management, resultant from a recent extensive review of the state-of-the-art [25] is the following: competence management concerns the way in which competencies in a corporation (of a group or individuals of the corporation) are organised and controlled. Competency (also called skill whenever related to humans) is a way to put in practice some knowledge, know-how, and also attitudes within a specific context. Competence management therefore has the prime objective to well define and continuously maintain the set of competencies according to the objectives of the corporation. A quite extensive survey of competence management systems and approaches has been conducted by Draganidis and Mentzas in [56].

Typical examples of competence management tools are Skillscape Competence Manager [26] and SkillSoft Skillview [27]. Table 2 summarises their most important features.

Skillscape Competence Manager	SkillSoft Skillview
Personal Skill Gap Analysis	Single assessments
Personal Development Planning	Gap analysis
Performance Planning	Individual and group reporting capabilities
Career Management	Skill-based searches
Organisational Skill Health Analysis	Multi-rater (360 degree) employee assessment
Predict Training Demand	

Skillscape Competence Manager	SkillSoft Skillview
Team Building and Internal Resourcing Succession Planning	Recruiting elements Total control over skill library, jobs and job profiles Skill weighting capabilities Employee resume posting and searching Employee free form posting and searching Employee scheduling

Table 2: Summary of typical competence management tools' features

Most of the aforementioned features are self explanatory. For detailed descriptions the reader is referred to [26] and [27].

3.2.3 Intelligent Requirements Assistants

Developers of complex software systems are challenged to demonstrate how a software system satisfies a set of customer requirements expressed as operational scenarios [28]. Scenarios are a key approach to eliciting and validating requirements. They provide concrete examples of system usage that can help in stimulating critical inquiry into systems requirements. However requirements that were elicited based on scenarios are validated with much effort and are often error prone.

Authors in [28] propose an "Evolutionary Requirements Analysis (ERA)" tool which applies evolutionary computing techniques to automatically select optimal combinations of human and machine agents in a system model to match non-functional requirements. The tool assesses the reliability, performance times and cost of different system models by executing many model variants, as evolving forms, with scenarios and different combinations of environmental variables. Better performing models are selected, to converge on an optimal solution.

Authors in [29] present a knowledge-based requirements assistant (KBRA) that is a component of the knowledge-based software assistant (KBSA). The idea behind KBSA is to create a knowledge-based life-cycle paradigm spanning software development from requirements to code and to formalise software practice so that computers can be used as active reasoning agents in developing software. They identify knowledge-representation issues associated with requirements acquisition and analysis, and note the three realms in which mechanisms operate to resolve knowledge issues: presentations, structured text, and evolving system description. They describe artificial intelligence techniques used to provide consistent reasoning processes for the intelligent assistant: inheritance of properties from generic object types, automatic classification based on discriminators indicating how to specialise instances, and constraint propagation for processing ramifications of requirements decisions and for supporting retraction when people change their minds.

3.2.4 Knowledge Based Program Designers

Program designers are tools that present their user with algorithms that match current requirements and specifications. These tools help transfer knowledge from the requirements

phase to the design phase. Algorithms are either generated with a machine learning approach or retrieved from the knowledge base by case-based reasoning [19].

Tools like CAESAR [30] use case-based reasoning for retrieving algorithms from the knowledge base by matching current requirements and specifications. This great help in code design comes with a shortcoming. CAESAR tries to match every possible case with the current requirements. Although the result might be close matches, it becomes cumbersome for the user to deal with so many examples. While CAESAR presents the user with all possible cases that can be reused, RT-Syn [31] tries to select one single possible algorithm. It looks into the algorithm database, choosing a likely candidate algorithm, and then makes design decisions based on the given constraints in requirements and specifications [32]. Designer Assistant [33] addresses three kinds of knowledge in its knowledge base:

- Expert knowledge of design with which most designers are not familiar.
- Impact knowledge (i.e. how characteristics of a design affect another area of software).
- Fault prevention knowledge (i.e. how characteristics of design could lead to a fault).

Tools or design environments like the Designer Assistant not only help transfer design knowledge which exists as folklore in organisations, but they also help organisations build their knowledge bases and increase the expertise level of designers and developers.

3.2.5 Knowledge Based Code Generators and Recommendation Systems

When coding to a framework, developers often become stuck, unsure of which class to subclass, which objects to instantiate and which methods to call. Example code can help developers make progress on their task [34]. Code generators rely on a previously acquired knowledge base (built from examples, or existing applications) that may or may not evolve, and strive to generate executable code. There are two kinds of code generation tools, using push and pull technology respectively [19]. Tools using push technology will automatically let the developer know about every possible opportunity of reusing previous acquired knowledge and components. On the other hand, in tools using pull technology, the user has to actually search for knowledge and reusable components.

An exemplary tool using push technology is CodeBroker [35]. CodeBroker queries a repository automatically after each comment or method signature written by a developer. The queries made to the repository are based on these comments and method signatures. To retrieve matches, a developer must write comments that explain the functionality of the software in terms similar to that of the repository code.

In CodeFinder system [36], the developer formulates a simple text query (using pull technology), executes the query and is then presented with a list of terms in the repository that are similar to those in the query. Depending on the terms and options selected by the developer, a different set of restrictions is presented to help narrow the search space to a specific class of examples of interest.

The Strathcona tool [34] is a plug-in for the Eclipse IDE which extracts the structural context of the code on which a developer is working when the developer requests examples. The server portion of the tool houses the example repository and selects examples to be returned using a set of structural matching heuristics. Authors in [34] consider that an appropriate example is a subset of one of the applications stored in the repository, consisting of a set of relevant classes and relationships. The developer is presented with a structural overview of

each example using a compact visual representation. The developer can access a rationale for why the example has been returned, as well as the source for the example.

There is a great variety of tools with similar functionalities such as KIDS [37], SINAPSE [38], CodeWeb [39], Component Rank [40], Reuse View Matcher [41], Automatic Method Completion [42], Hipicat [43] and RASCAL [44] to name just a few.

3.2.6 *Smart Code Analysis Tools*

Code analysis tools can offer substantial help during software testing and quality assurance activities. Analyzing code requires expert knowledge about the quality of the written code and good programming style. This knowledge is captured in a knowledge base and integrated with tools like OGUST [32] that analyze the code for quality and good programming style.

Style checkers such as PMD (<http://pmd.sourceforge.net>) turn up poorly named variables, duplicated code, and many other deviations from coding conventions. Bug detectors like FindBugs (<http://findbugs.sourceforge.net>) spot common reliability problems such as dereferencing null pointers, infinite recursive loops, and broken idioms that do not achieve what the author intended.

A system for detecting redundancies in source code is R^2D^2 [27]. R^2D^2 identifies redundant code fragments on large software systems. For each pair of code fragments, R^2D^2 uses a combination of techniques ranging from syntax-based analysis to semantics-based analysis, that detect positive and negative evidences regarding the redundancy of the analyzed code fragments. These evidences are combined according to a well-defined model and fragments sufficiently redundant are reported to the user. R^2D^2 explores several techniques and heuristics to operate within reasonable time and space boundaries and is designed to be extensible.

Another approach is program slicing, which was first introduced by Weiser in 1979 [46]. A definition in [47] for program slicing is the following: “*a decomposition technique that extracts from program statements relevant to a particular computation*”. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest referred to as a slicing criterion. Typically, a slicing criterion consists of a pair $\langle p, V \rangle$, where p is a program point and V is a subset of program variables. The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion C are called the program slice with respect to criterion C . The task of computing program slices is called program slicing.

Slicing was first developed to facilitate debugging, but it is then found helpful in many aspects of the software development life cycle, including program debugging, software testing, software measurement, program comprehension, software maintenance, program parallelisation and so on.

3.2.7 *Software Maintenance Tools*

Oman in [48] defines software maintenance as “*the Software Engineering activities pertaining to non-essential enhancements in software, adapting the software due to changing environmental requirements, and the removal or mitigation of faults*”. Software maintenance efforts and research focus on prevention and removal of software faults throughout the Software Engineering life cycle, including issues that arise from modification of software after delivery.

Authors in [49] describe a novel approach in providing automated re-factoring support for software maintenance: the formulation of the task as a search problem in the space of alternative designs. Such a search is guided by a quality evaluation function that must accurately reflect re-factoring goals. Based on this approach they have built a search-based software maintenance tool. CODE-Imp applies automated re-factorings to a program in order to move through the space of alternative designs and search for those of highest quality, based on a given design quality function. The effectiveness of the search can be measured in terms of the change in quality function, but the effectiveness of the approach itself can only be judged in terms of the actual changes made to the program and to what extent it is more maintainable than the original. For this reason, choice of design quality function is a key facet of this work.

Another approach proposed by Mens et al. in [50] is a lightweight abstraction of intentional source-code views as a means of making the conceptual structure of existing software systems (which is often implicit or non-existing in the source code) more explicit. An intentional source-code view (as defined in [50]) is a set of related program entities (such as classes, instance variables, methods, method statements) that is specified by one or more alternative descriptions (one of which is the 'default' description). Each alternative description is an executable specification of the contained elements in the view. Such a description reflects the commonalities of the contained elements in the view, and as such, codifies a certain intention that is common to all these elements. In addition, all alternative descriptions of a given view are 'extensionally consistent', in other words, after computation they should yield the same set of elements. The computational medium in which the intentional views are described is a declarative metaprogramming language. The practical usefulness of intentional views to aid software maintenance is described in [50] in the context of two case studies that Mens et al. conducted.

3.2.8 Wikis

According to [51], Wikis are simple to use, asynchronous, web-based collaborative hypertext authoring systems. The general consensus is that a Wiki is a collective website where a large number of participants are allowed to modify any page or create a new page using their web browser.

Wikis are also able to provide assistance in the Software Engineering process. Chau and Maurer in [52] propose a wiki-based experience repository (MASE) that utilises both informal and formal knowledge representations. The need of knowledge sharing tools to incorporate not only codification-oriented repository technologies but also those that facilitate communication and collaboration among people, and to support not only structured but also unstructured knowledge representation is obvious. An informal knowledge authoring tool such as MASE is used for sharing content for problem understanding, instrumental, projective, social, expertise location, and content navigation purposes. Semantic media Wiki¹⁶ is using semantic technologies to represent knowledge and can therefore be used to assist MediaWiki (non-semantic)¹⁷.

¹⁶ http://ontoworld.org/wiki/Semantic_MediaWiki

¹⁷ <http://www.mediawiki.org/wiki/MediaWiki>

A more detailed review of Wikis and especially semantic Wikis will be presented in D7: “Report describing state-of-the art in SE Knowledge Desktop”.

3.2.9 Collaborative Tools

Software engineering is a predominantly collaborative activity. Typically multiple teams of people develop and maintain successive versions of a range of products in parallel. Surprisingly, tools to support synchronous or real-time Collaborative Software Engineering (CSE) are still restricted to minor tasks for specific Software Engineering purposes [53].

Cook and Churcher [53] propose the CAISE architecture to assist in the construction of new tools to support the real-time development of a collaborative software project. The CAISE architecture, allows isolated programmers to work collaboratively without sacrificing communication. CAISE-based tools achieve this by keeping all programmers within a group synchronised in real-time, at the same time providing customisable user awareness and project state information to the individual tools. The CAISE architecture provides an infrastructure with the potential to support the entire Software Engineering process. CAISE tools can be constructed that provide more than just the shared editing of basic software artefacts. Collaborative compilation, testing and debugging of software projects are also possible to implement using the services of CAISE. Comprehensive inter-developer communication facilities can also be constructed.

The Augur system [54] can be viewed as an example technique to look at software development as a system of evolution. The Augur system simultaneously visualises the structure of a software system (i.e. artefacts) and the structure of the development process carried out by developers (i.e. developers and the community). Augur visualises the result of call graph analysis and networks of contributors to a project, relating those who worked together on a single module. By looking at how developers worked together on what parts of a software system, a user of Augur could tell how relationships between artefacts (software system module structures) and developers have changed over time, including phenomena such as types of projects, the different roles undertaken by different developers, how such roles shift between core and periphery, how authorship changes, and what patterns of stability and changes are observable. Augur currently supports ways to view the structural changes from an objective standpoint, providing ego-centric individual viewpoints, for instance, from a particular developer’s point of view.

Another approach to CSE is using hybrid representations of online activities. Authors in [55] have developed a system to observe and analyze collaborative activities in online groups merging three data sources: not only communication patterns (i.e. email traffic) but also topical and material relationships. Each component in this hybrid network allows easy access to the raw data, so that analysts can examine the qualitative information behind the structure of the network. They show how the simultaneous visualisation of heterogeneous data reveals collaboration patterns that would not have been visible using social networks exclusively.

4 KNOWLEDGE REPRESENTATION

So far we have analysed various methodical approaches as well as systems for managing, i.e. capturing and sharing knowledge in the field of software development. A crucial aspect for such systems is the formal representation of knowledge. In fact, knowledge representation is one of the biggest Knowledge Management challenges. Knowledge representation deals with the ways people represent their own knowledge, similarities and differences between these ways, as well as smart methods for communicating this knowledge to computer systems, i.e. storing, manipulating and retrieving it. Thereby, the essential difficulty is that knowledge is not a physical object but an abstract product of the human consensus, which rarely exhibits definite properties.

In the present section (Section 4) we discuss different approaches for representing software developers' knowledge from two major perspectives. First, we present the state-of-the-art of representing knowledge about software systems, being under development. This affects mainly "what should be done?" and "how could it be done?". For that reason, we examine and compare common practice of system modelling as well as the emergent ontology-based software development. Then, we look at knowledge produced during the development process, resulting from different collaboration efforts. This perspective concerns Software Engineering decisions, which use this knowledge as an input and further advance it after the decision is taken. Development knowledge is more about "why is it done in such way?", i.e. the rationale behind development decisions.

4.1 Representing Knowledge about System Models

Software engineering involves a number of development activities, including requirement elicitation and analysis, design, implementation, testing, maintenance etc. During these activities developers always produce artefacts that describe different system aspects. These artefacts are called system models. For instance, a use-case specification is a system model describing one system functionality. A system architecture is also a system model describing system components and their relationships. Pieces of code or web service specification are system models as well, describing the system in a specific machine readable language. Of course many different languages, techniques and tools are used to represent system models. In this section we will first focus on models and notations as a settled mechanism for representing human oriented knowledge about the software. We will then describe machine oriented knowledge representation techniques, especially the emergent field of ontology based software development.

4.1.1 Models and Notations

Bruegge and Dutoit define Software Engineering among others as a modelling activity, where "*software engineers deal with complexity through modelling, by focusing at any one time on only the relevant details and ignoring anything else*" [57].

By means of well defined notations, modelling enables software engineers to precisely and concisely articulate complex information about the system. Especially in collaborative and distributed environments developers need semantically defined notations in order to represent their knowledge about the system and be able to communicate and discuss it [58, 59]. Depending on the current development activities, the role and expertise of the developer,

the articulated information can vary from a simple idea to detailed knowledge. For instance, during analysis of the system requisitions, requirement engineers often specify the acquired knowledge about the application domain in the analysis model [60]. During system and object design, developers make intensive use of diagrams to fill the gap between analysis and solution objects in a highly creative process. Models are a perfect toolkit for rounding out ideas about reusing patterns or re-factoring classes, e.g. generalising a student and a professor class to a superclass person.

Software systems are knowledge-intensive, complex and incorporate diverse aspects. Modelling helps in externalising, representing and communicating this knowledge. In conclusion, modelling in Software Engineering is both a knowledge acquisition and knowledge sharing activity. Software development can even be considered as a knowledge representation activity, since no real production takes place in developing software. Software engineers specify during development their knowledge about the system in both human and machine-readable languages, in order to respectively communicate, collaborate and document the system as well as to produce the system functionality in computer systems.

Indeed, modelling is not new to Software Engineering. At least since P. Chen introduced his Entity/Relationship (ER) model and diagrams, conceptual modelling is considered as a key activity for database development. Shortly thereafter, software development communities recognised the importance of modelling. As a result ER techniques have strongly influenced most development methodologies, languages and tools. In order to meet the engineering and technical progress in this field, software development communities have modified, re-defined and extended ER diagrams, e.g. by inheritance mechanisms or multiplicities.

The Unified Modelling Language (UML)

With the emergence of object-oriented techniques, new aspects of behaviour modelling were born. Many proposals of modelling notation were made until the end of the past decade when UML came out as a result of the unification of the three most important ones, namely Object Modelling Techniques (OMT) [61], Booch [62] and Object Oriented Software Engineering (OOSE) [63].

UML is currently a widely used industry standard. It offers a wide diagram catalogue in order to capture knowledge about three different system aspects:

- **Functional aspect:** UML offers use case diagrams to describe functionality of the system from the user's point of view.
- **Structural aspect:** amongst others class, object and deployment diagrams can be used to describe system concepts and their logical relationships from different levels of abstraction.
- **Dynamic aspect:** interaction, activity and state diagrams can be used by developers to represent behavioural aspects of the system.

The structure of UML diagrams is reflected in Figure 5 according to the Object Management Group's specification [76]. Class, component, use-case, state, activity and sequence diagrams are the most common ones. These diagrams are largely sufficient to model the majority of software systems' properties. The other concrete diagram types (highlighted in grey on the Figure) are helpful for modelling specific system aspects, e.g. timing diagrams for embedded software such as the control for fuel injection system in an automobile.

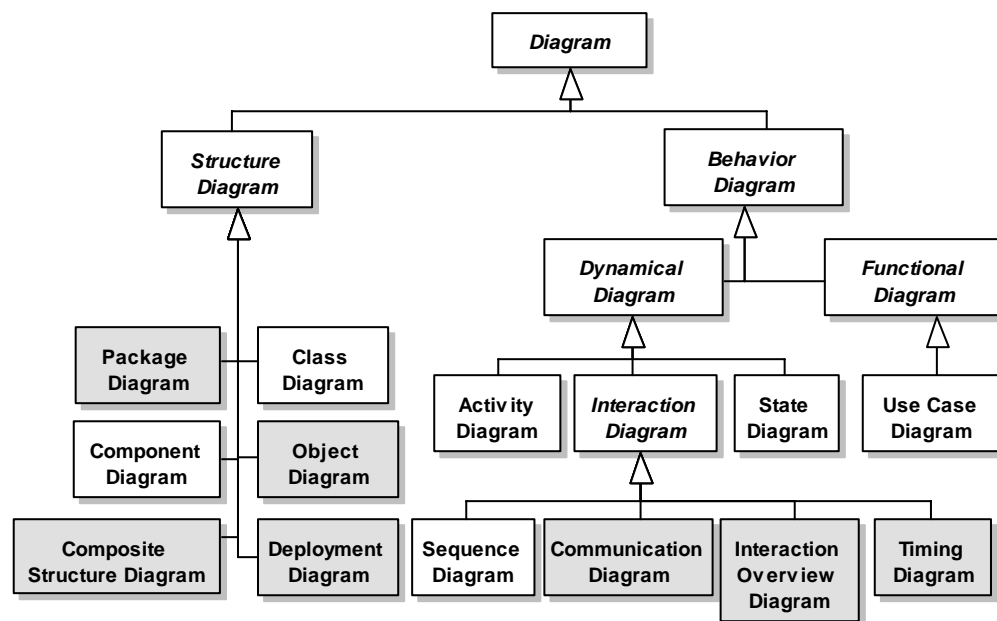


Figure 5: Taxonomy of UML diagrams represented as a UML class diagram

A detailed insight into UML is given in [64]. An introduction to modelling with UML applied to the different activities of Software Engineering can be found in [57].

UML 2.0 and MDA

In UML 1.5, action semantics were introduced, providing developers with a complete, platform-independent way for specifying actions in their models. The main goal of action semantics was to make UML modelling executable, i.e. to allow developers to test and verify their models and to generate 100% of the code if desired.

Action semantics led to one key feature of UML 2.0: to allow a very precise definition of the semantics of the models, in the sense that developers can directly translate UML models into programs. In fact, this is a part of a much more general vision of the Object Management Group's (OMG) Model Driven Architecture (MDA)¹⁸, which is shifting developers' focus from the programs towards models.

This leads to the first major requirement of UML 2.0, namely providing support for the Model Driven Architecture and model-driven development. That requirement demanded more precision concerning the semantics of UML than there was originally in the first versions. MDA and UML 2.0 support the ability to meet higher levels of abstraction, overcoming obstacles of underlying implementations, programming languages and platforms details.

The second fundamental requirement of UML 2.0, originating from the fact that software systems become increasingly larger, more complex and diverse, is the ability to model very

¹⁸ <http://www.omg.org/mda/>

large heterogeneous systems. Developers should be able to describe the different aspects of such systems succinctly.

Modelling is one of the most important and specific to software development tools for Knowledge Management, since models are much closer to the way developers design applications than a piece of code or a textual description. One can consider system models as overlapped interrelated knowledge islands each focusing on one system aspect and ignoring irrelevant details. Together models build a coherent and complete knowledge base about the system, which is easier to acquire from and to share between development teams.

4.1.2 *Ontology-based Software Engineering*

In the last decades ontologies won on popularity in several fields of computer science. The most notable usage of ontologies is in the fields of artificial intelligence, multi-agent systems, and Web technologies. The term ontology is borrowed from the philosophy, where it stands for a systematic account of existence. Deviating from its original meaning, in computer sciences ontologies are considered as a network of relationships that are self-describing and used to track how concepts are related to each other. Gruber [77] for instance, defines ontology as “*a formal and explicit specification of a shared conceptualisation*”.

In Software Engineering conceptualisation plays a major role, e.g. during requirement analysis when analysts extract problem domain concepts from the requirement specification. A second example coming from the design activity is re-using a design pattern as a well defined solution concept. Since building ontologies is about specifying concepts, ontologies could and should play an important role in Software Engineering.

However, ontologies do not have a major role in the Software Engineering community as it is the case with models. Up to now, ontology engineering and Software Engineering presented two parallel research areas with different communities of interest. Until recently and with the emergence of the semantic web, leading research organisations such as the WWW Consortium and OMG took initiatives [65, 66] aiming to better integrate both areas by defining ontology development platforms and investigating best practices for ontology driven architectures.

For instance OMG defines the following statement as one vision of its current activities: “*bringing together the expertise in the semantics of software with the expertise for knowledge representation, consistency checking, and knowledge-processing, in order to leverage each other’s technologies*”. Recent efforts of the OMG resulted in a detailed comparison between the Web Ontology Language (OWL) and UML as well as a mapping mechanism between them. This could become a major milestone for bringing software- and ontology- engineering, their communities, tools and standards much closer to each other.

Indeed ontologies and System Models often have many similarities. Hesse [67] compared ontology engineering and structural modelling and found out that both approaches have in common the goal of representing the relevant “things” of an application area and their interrelationships. For defining an ontology and a structural model the same concepts and languages may be used. In both fields there are “syntactical” and “semantical” aspects: The first encompass the naming, classification and structural description of concepts and their relationships. Parts of the “semantics” are further conditions, constraints and associations. However, there are differences as well. The most important difference is the scope. Models are usually used for one particular project. Their concepts do normally not refer to areas beyond the project scope. Differently, ontologies supply a much bigger group of clientele, or

even computer programs, which do not have to belong to the same project or even to the same organisation. As a result ontologies always try to represent universally valid truth (knowledge) about a restricted domain. They encompass future projects and developments including potential, possibly still unknown ontology users. For a concrete comparison between data models and ontologies the reader is referred to [67].

Moreover, some authors have already demonstrated how Software Engineering can benefit from ontologies. Knoblauch [68] demonstrated some initial thoughts on a software architecture and methodology in an ontology-driven development. On the one hand this architecture supports sharing domain knowledge to the public; on the other hand re-using other available knowledge at run time is also supported. Experimentations with a generic, ontology-based search engine demonstrated that the use of ontologies within software architectures offers domain independent semantic search functionality without the need to programmatically adopt the search functionality when the structure of information is changing within the domain [69]. Scott and Padmapriya [70] focused on deploying ontologies to represent, share and re-use solution domain knowledge. They demonstrated a technique to formally represent design patterns and allow machine composition of patterns into coherent design solution.

Even if first researches in ontology-based Software Engineering are promising, this field is still in an early experimentation phase. Perhaps this is the reason why no concrete industrial application has been introduced to the community yet. However, it is expected that this field becomes one of the most agile in the near future.

4.2 Representing Development Knowledge

Many researches have noted the importance of collaboration in software development [58, 59, 71]. During collaboration developers discuss and resolve issues, annotate or comment system models or just share their experience. Kraut and Streeter [72] noted that formal communication (e.g. planned meetings, demonstrations, inspections) is useful as routine resource coordination. However, informal communication (e.g. e-mails, phone calls, hallway discussions) is required to handle uncertainty and risks, which are typical to software development. These researches amongst others stressed out that informal communication is critical for rapidly disseminating implicit knowledge in software development, in particular in the emergent agile development methodologies. One of the oldest Knowledge Management approaches that addresses the collaborative and debate-based nature of software development is managing design rationale. In this section we first introduce rationale management in software development. Then, we investigate different approaches related to capturing and sharing rationale information and demonstrate how these result in development-related knowledge transfer in Software Engineering.

4.2.1 Rationale Management

One of the most valuable knowledge created during software projects, but usually not explicitly captured is rationale. Rationale is the justification behind decisions. Rationale usually includes:

- Issues that arise during developing, maintaining and using software and need to be addressed.
- Alternatives that were considered to address issues.

- Criteria as desirable qualities that the selected solution should satisfy.
- Arguments illustrating the debates developers went through to reach the decision.
- Decisions as resolution of an issue representing the selected alternative according to the criteria used for evaluation and justification of the selection.

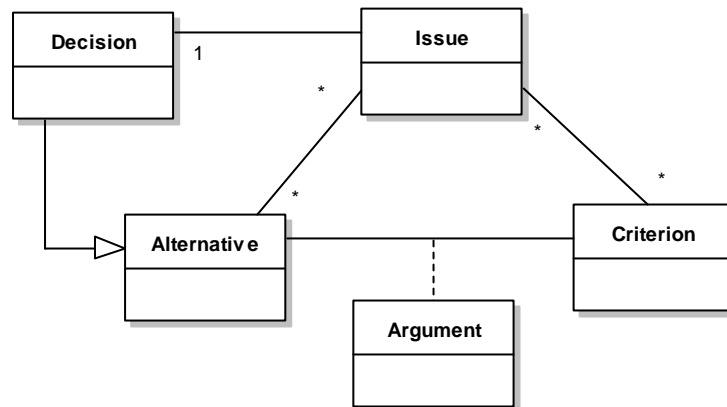


Figure 6: Rationale objects

Since decisions should be based on detailed knowledge about the issues, alternatives and criteria, capturing rationale normally leads to capturing knowledge.

Particularly if changes in the problem or the solution domain occur (e.g. requirements change or a new technology arises) rationale becomes a critical knowledge, making it very hard for someone that was not involved in the decisions to understand why the component is designed the way it is. A detailed introduction to rationale and its management can be found in [57], Chapter 12.

Unfortunately, rationale is one of the most complex information developers generate. For that reason, externalising and representing rationale knowledge is very exhausting. In most of the cases, issues, alternatives and arguments for a decision are communicated informally and remembered by individuals. Over time this information degrades and gets lost when staff leaves the organisation. In addition, capturing information about alternatives that were considered, but not implemented, presents a major challenge. Managers as well as developers always scrutinise the profitability of such a major investment of time and resource. Indeed this challenge is not specific to rationale management. It is more an issue of Knowledge Management in general.

4.2.2 Capturing rationale

The most famous approaches for rationale representation are briefly introduced below:

Issue-Based Information System IBIS: The idea of capturing rationale is much older than Knowledge Management itself. It goes back to the seventies as Kunz and Rittel presented their issue-based information system IBIS [73]. Indeed, IBIS was not a software system but a way of modelling argumentations [73]. IBIS implemented the Rittel's argumentative approach to design problems that cannot be solved algorithmically, but, rather, have to be addressed through discussions and debates.

Question, Option and Criteria (QOC) is a further method for argumentative design space analysis [79]. While IBIS is used to capture rationale as it occurs during the design process

(for example during team meetings), QOC rather aims to explore the space of all possible options for the given issue and tries to explain why a certain option was chosen over others. The representation focuses again on three main concepts. Questions represent main issues for structuring the design space and options are the possible answers to these questions. New questions may arise from some options. Criteria are used by the designers to evaluate individual options which can either be assessed positively or negatively against these criteria. Additionally, arguments can either object to or support any question, option or criterion.

Decision Representation Language (DRL) [80] represents an extension of IBIS by adding nodes to capture design goals and procedures. Providing a finer level of granularity, it enables answering a larger range of questions that might arise in different development phases. The main drawback of DLR is its complexity and the efforts spent on structuring the captured rationale.

NFR Framework [20] is an approach for representing and tracking non-functional requirements for each decision, evaluated alternatives as well as interactions between them. Unlike the argumentative methods presented above, the NFR Framework is specific to requirement engineering. Non functional requirements are considered as goals to be met. To reduce decision complexity, goals can be recursively refined in sub-goals. The refinement process results in a goal graph, including AND and OR relationships between the parent goal and the subgoals. AND decomposition implies that all subgoals have to be met to meet the parent goal, while in OR decomposition only one subgoal suffices. An interesting concept of the NFR Framework is the reuse of goal sub-graphs from one system to another. This enables developers to pick groups of solutions that were explored and evaluate them against new criteria.

In fact, especially for design rationale, there exist many proposals for approaches, systems and representation of this knowledge. Regli et. al [74] provide an excellent survey of design rationale systems and concepts. They find out three main challenges for such systems:

- Technical challenges: design rationale technologies need to be designed specifically to suit the needs of the organisation in which they will be used.
- Design challenges: they must support both formal and informal knowledge, making the system flexible enough so that broad content types are supported. They must support multiple levels of organisation of content and design systems so that knowledge can be structured at any time after it is entered in the system.
- Automatic retrieval challenges: automatic retrieval is appealing but it is imperative to find the boundary across which systems become intrusive rather than helpful.

Recent researches demonstrated the importance of rationale for the different software-life-cycle activities [57] 75, 78], introducing the term “Software Engineering Rationale”. As a matter of fact rationale stakeholders are not limited to designers. Requirements stakeholders, system maintainer, constructors and testers are also concerned about different rationale categories (see Figure 7). For example, requirement rationale [75] uses options for managing the evolving requirements. Requirement engineers might propose, evaluate and refine different options for implementing a change request. In such a way, an impact analysis of different options resulting from change can be precociously carried out based on the captured rationale graph.

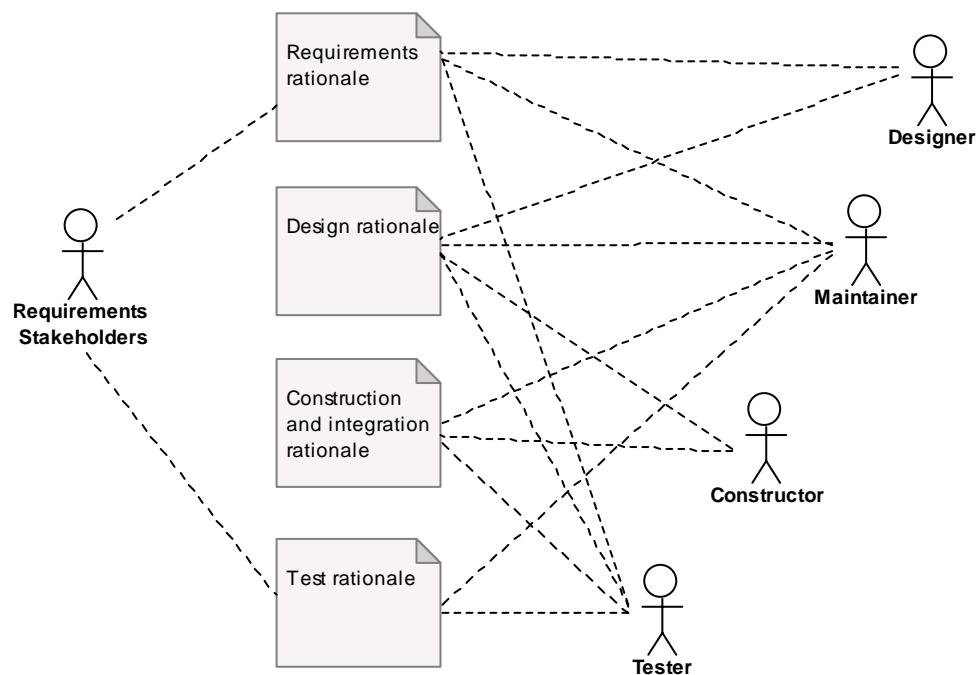


Figure 7: Rationale use in Software Engineering (from [78])

Rationale can be useful for all decisions during software development [78], particularly for the following purposes:

- Acquisition and supply: e.g. during supplier selection to justify how the components satisfy the acquisition requirements.
- Engineering: rationale essentially improves the knowledge sharing between the different engineering roles. A software maintainer, for instance, is able to understand why a piece of code is written in a particular way.
- Operation: not only the support team but also the customer themselves can be assisted in understanding how the system will behave by rationale knowledge captured during engineering.
- Support: captured rationale can be used to generate documentation, explain why specific processes were not executed or why certain artefacts were not provided.
- Management: as management needs to understand the factors that have led the project to a specific situation, rationale is an ideal way of tracking such factors. Capturing rationale also helps in exploring the whole solution space and identifying the risks and the opportunities related to each option.
- Reuse: As demonstrated above, rationale specifies the context in which an artefact is reusable and in which way it can be reused.
- Process improvement: rationale is definitely helpful in evaluating, understanding and accepting specific process steps.
- Resources and infrastructure: rationale is a valuable knowledge especially for newcomers to understand the current situation and learn from previous experiences.

5 EMPIRICAL RESULTS

The subject matter has been mostly of theoretical and technological nature so far. However, when studying Knowledge Management in the context of Software Engineering and especially collaborative software development, a grasp of the real world situations is required. By scientifically analyzing practices in real-world settings, problems can be identified and measures can be derived to evaluate success.

Empirical investigations are rare in computer science in general [104, 106], and the same applies for studying developers and team habits. However, with the advent of global and open source development and through the availability of large amounts of metadata from development repositories, there have been a rising number of efforts in this direction.

In the context of TEAM, there is a special interest in the area of sharing of knowledge in distributed development teams, which is tightly interwoven with communication and coordination aspects. Thus, we will draw from various studies in distributed software development, which in their majority are not dealing with the notion of knowledge sharing explicitly.

In the present section (Section 5), an overview of empirical research approaches in these areas, as well as of the variables and metrics used to measure success and failure of knowledge sharing efforts, is provided. Since several “spheres” can be identified in knowledge sharing, the overview is grouped by the unit of analysis, starting by presenting work on individual developers, proceeding with team and distributed development settings and concluding with some final remarks relating to Open Source development.

5.1 Personal sphere

Nowadays, few software projects can be overseen and tackled by a single engineer. Even when creating a small application, the developer has to rely on frameworks and toolkits created by others. However, the developer remains the smallest unit of analysis in Software Engineering and is, consequently, necessary to study the work behaviour of individual developers in order to characterise information and knowledge needs.

In 1994, Perry et al. [100] did a first investigation of how developers spend their time. They collected data from 13 developers from four separate groups inside one company. Their activities were recorded on a daily basis in a time diary for the period of one year. The diary form was created based on principal activities that were identified in a pilot study. The recorded data was verified by observing a random sample of five developers for five days. While the analysis of the diary data does not reveal what activities directly involve communication and information gathering, it is stated that coding and non-coding tasks (like design or testing) both make up approximately fifty percent of the time spent. However, the observation part of the study revealed, that there was a large amount of unplanned, informal communication with an average of 75 minutes per day. The median value for personal contacts per day was at seven persons.

Results of a similar study were reported by Singer et al. [103] in 1997. Using a free form, web-based questionnaire, they first identified a total of 14 categories of work activities. Based on those, they performed a longitudinal study observing one single developer for 14 days and a second study monitoring eight developers for one hour each. Both studies, the actual occurrence of the activities was documented as well as the number of single events

belonging to each activity category. Among the categories, documentation, management activities (including meetings), consultations, note taking and searching information made up information management tasks.

In a more recent study, LaToza et al. [96] investigated the activities of developers with a special emphasis on coding-related tasks. A list of nine basic categories for those tasks has been set up. Among those, “communication” was the only knowledge-management related category. Data was gathered by performing two surveys, each with nearly 200 participants and eleven interviews. In average, developers answered to spend about 10% of their work communicating. Out of this time, they spent about 40% in meetings (more unplanned than planned) which they rated as quite effective (about 5 on a 1-7 scale with 7 meaning “highly effective”). E-Mail was also heavily used (25%) and perceived as similarly effective. Less time was spent using bug databases and documents (both around 5-10% of the time) while instant messaging and phone calls were rarely used. In general, e-mail was blamed for the long response times and possible misinterpretations since detailed e-mails are hard to write. Thus, they were primarily used for issues of low-priority and those involving many people.

Study	Research question	Method	Knowledge-related activities
People, Organisations, and Process Improvement [100]	How do developers spend their time during the SE lifecycle?	Time diary and observation	Documentation, Unplanned informal communication
An examination of Software Engineering work practices [103]	What activities do software engineers perform?	Multi-Method: web-questionnaire, observation and log analysis at one company	Search for information, Documentation and note taking, Communication (personal consulting and meetings)
Maintaining mental models: a study of developer work habits [96]	How do developers spend their time with coding tasks?	Two surveys and eleven interviews at a company	Communication; preference for face-to-face communication; many interruptions; lack of documentation; people are major sources of information

Table 3: Research in individual developers’ work practices

The general setup of the studies described is summarised in Table 3. Although starting with quite similar research questions, the studies follow different approaches in data collection such as questionnaires, observations and interviews. While all studies do not explicitly focus on (personal) Knowledge Management issues, the categorisation schemes used for recording activities cover some of those aspects. Most notably, all of the described studies investigate a single company only. Together with the differences in the categorisation schemes used, this makes it very hard to generalise conclusions for software developers in general.

5.2 Team sphere

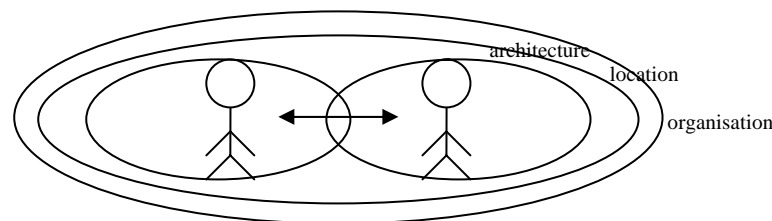


Figure 8: Relationships between developers in a team

As already mentioned, large systems can not be constructed by a single developer. Work has to be split between developers, resulting in a dependency relationship between them (see Figure 8). Thus, besides information for their concrete implementation tasks, individual developers need further information about the overall development process and system architecture, i.e. information to coordinate with the other team members. These information needs will be described by presenting several works in that area. At this section, the focus will be on aspects of collocated teamwork. Special issues of distributed work settings will be covered separately in Section 5.3.

5.2.1 Exploratory studies

Team software development is a rather old issue from a practical point of view. Early work on those issues, stimulated by the “Software Crisis” has primarily dealt with technical issues such as modularisation and component-based development, starting with the work of Parnas [99]. Although some authors were aware, that the success of team development is also depending on human factors [88], the scientific study of team work practices is rather new. Since there were few theoretical frameworks in place, authors started with exploratory studies, guided by very general research questions. Those questions were about what problems are encountered in team development and which best practices are adopted to deal with them (c.f. Table 4). From a methodological point of view, the studies are either pure experience reports or case studies, based on qualitative (interviews) and quantitative (questionnaires, log analysis) data collection techniques.

Investigating the existing KM practices and activities of teams in two Australian software companies, Ward and Aurum [107] found out that the primary form of knowledge exchanged in development teams was tacit knowledge. Accordingly, the major source of information reported by the developers was their communication with colleagues.

However, personal communication is a limited resource and becomes not feasible if project size increases. Cusumano [87] explored how Microsoft deals with this problem. He found out that they tend to reduce the amount of required communication by freezing architectural interfaces and standardising tools and processes. Furthermore, architectural entities and development teams were aligned, so that information can be exchanged easily.

Chau and Maurer [86] describe how asynchronous (Wiki) and synchronous (Instant messaging) Knowledge Management tools are used in a team setting. They found out that the paradigm of decentralised contributors adding unstructured knowledge improves knowledge sharing in contrast to structured knowledge provided by a centralised team. They also

discovered, that developers preferred asynchronous over synchronous media for exchanging knowledge.

The aforementioned studies support the findings from the personal sphere that personal communication is the most important medium for knowledge sharing. Thus, organisations try to align work groups and technical architectures to create a broad communication space, supplemented by standardised processes and tool environments to reduce coordination needs.

5.2.2 *Specific investigations*

Based on these experiences, researchers turned towards more concrete research questions. Methodologically, those studies are more sophisticated than the exploratory works. Beneath basic theoretical hypotheses, the investigated sample is larger – either by the number of projects or organisations.

Curtis et al. [85] investigated how human factors influence software development in terms of quality and productivity. They identified multiple layers in the organisation ranging from individual, team and company level up to the business milieu. While they found out that unsurprisingly there is less frequent communication among the levels, communication changes also in terms of context and channel. Information is filtered at boundaries and loses context information from the transmitters' side.

Walz et al. [108] looked at differences in knowledge acquisition and sharing among group members. Towards that direction they visited and video-taped 19 design meetings in one large software research and development project. They found out that sharing knowledge was primarily useful to build a common mental model of the system under development. Also, lots of learning happened in the course of design discussions. However, much of the knowledge created and discussed was not captured and thus, lost for people not participating in these discussions.

Kraut and Streeter [97] investigated 65 projects regarding their differences in coordination techniques. They found that informal communication is especially necessary for coordination in projects in a stage of high uncertainty. The results also support developers' preference for contacting collocated colleagues. An important variable was the ease of getting information. People that were difficult to access were less frequently contacted, even if they could provide valuable information.

5.2.3 *Summary*

While most of the described studies do not conceptualise Knowledge Management activities, they are clearly driven by those issues. Basically, the studies are dealing with phenomena (communication) and concrete measures to improve it (e.g. modular architecture or common practices). However, the underlying knowledge issues are architectural knowledge (also referred to as “shared mental models” of the system) and knowledge about certain work practices (e.g. coding conventions).

Study	Research question	Method	Major results
How Microsoft Makes Large Teams Work Like Small Teams [87]	How does Microsoft make large teams work like small ones?	Experience report	Reduce communication needs by best practices (freeze cycles, stabilise design), collocation, common coding styles, standardised tools, architecture

Knowledge Management in Software Engineering - Describing the Process [107]	Find out Knowledge management practices and activities in Software Engineering	Two case studies (12 structured interviews and qualitative questionnaire) Australian companies – two projects each	Most knowledge is tacit; source: communication/ colleagues; Currently applied tools and methods are inadequate
A case study of wiki-based experience repository at a medium-sized software company [86]	EXP How do users collaborate? What types of knowledge? Do they self-organise?	Case study; logging to Examine MASE wiki (codification and personalisation) in empolis	Developer prefer asynchronous tool. Greater need for unstructured knowledge.
A field study of the software design process for large systems [85]	How do human/social factors affect SD; how do problems in designing SD affect quality and productivity	Field research, interviews in 17 large projects → Multiple levels of analysis Spread of app and domain knowledge?	Communication changes context and channel; developing is a learning, communication and negotiation process; written documents did not capture decision dialectics
Coordination in Software Development [97]	Does coordination become more difficult with rising project size and complexity? What are the roles of formal and informal coordination	Coordination practices in large SW systems Survey, N=65 projects in one company	Importance of Informal and formal communication differs in project situation; ease of getting information is crucial.
Inside a software design team: knowledge acquisition, sharing, and integration [108]	How do group members acquire share and integrate knowledge? Do the levels of participation differ across members?	R&D project at MCC, 19 video-taped design meetings	Design teams share knowledge to build mental models; context-sensitive learning important; much information is never captured; increase application domain knowledge in team, allocate time for learning, role of conflict Three topics of discussions in meetings: background knowledge, requirements, design approaches

Table 4: Research in team software development

5.3 Distributed development

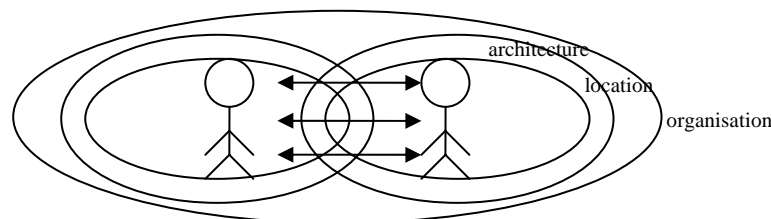


Figure 9: Relationships between developers in a distributed team

While the preceding section reported on studies of development teams in general, there is a growing body of research focussing on the specific challenges of distributed team work. In opposite to collocated work, such a setting is characterised by teams consisting of members separated by geographic and/or (inter-)organisational boundaries¹⁹ (see Figure 9). With the advent of flexible project teams in large software companies and offshore contracting, this form of work is increasingly adopted.

However, in those settings, most companies try to minimise the need for coordination by providing clear boundaries and separating work as much as possible. Collaboration and knowledge sharing problems are thus less regarded from a personal or group perspective, but with a focus on certain work processes, such as bug resolution.

5.3.1 Exploratory studies

Like in team development, research started with exploratory studies to identify the most pressing problems and possible solutions. Accordingly, the research questions were as broad as how distributed teams communicate and coordinate. Since the research setting by definition involved at least two sites, the case studies are mainly based on interviews.

Paasivaara [98] explored why people communicate in distributed projects and what structures exist to support them. Four important communication needs were identified: problem solving, information and monitoring, relationship building and decision making. To support them, the analysed organisations used three types of measures: organisational structure (i.e. clear definition of roles), synchronisation of processes and project level coordination (e.g. steering committees and weekly meetings).

Further research about how distributed teams coordinate was carried out by Ramesh and Dennis [101] based on media richness theory. They found out that coordination changes during the project lifecycle. In this course, team performance increases, when communication media matches the task needs. Rich media can help to decrease the overall communication needs.

Herbsleb and Grinter [90, 91] turned towards coordination problems of distributed teams. By interviewing software engineers working in remote groups in the UK and Germany, they identified some core problems. A major communication barrier was the lack of unplanned contact between group members. Thus, it was difficult to know whom to contact with questions relating to the remote site. Also, contact was difficult due to high cost and a lack of trust among the sites. Besides reducing coordination needs, best practices to overcome those barriers were frequent travelling and the appointment of liaison people.

5.3.2 Specific investigations

Although the research setting is quite complex, there are also some more specific studies dealing with distributed development. Espinosa et al. [89] investigated how shared mental

¹⁹ Different time-zones can be considered a third criterion. However, pure “follow-the-sun” approaches are a sub-topic of distributed development with rather limited practical relevance due to the enormous additional coordination challenges [82, 105].

models, work familiarity and distance affect coordination. They defined shared mental models as “*organised knowledge that members share about things like the task, each other, goals and strategies*” and work familiarity as “*the specific knowledge that members have about aspect of the workplace*”. Testing hypotheses by interviews, surveys and studying archival data revealed that the positive effect of shared mental models and work familiarity is stronger for distributed teams.

Herbsleb and Mockus [92, 93] studied the resolution process of modification requests in a distributed project. Therefore they analysed CVS repositories and survey data. Their hypothesis was that modification requests involving people from remote sites take longer to resolve than those involving people from a single site. Their data support that distributed work suffers from delays due to communication overhead, since generally more people were involved in requests spanning multiple sites. Developers tended to prefer informal communication because formal communication does not react quickly enough.

5.3.3 Summary

While the general points made for team development also apply for distributed teams, they are even more critical, since knowledge sharing gets more difficult with increasing distance. Some channels, like face to face meetings, become very expensive or even impossible to use. Since this may already happen when collaborating with colleagues at another floor, companies must strive to create “virtual 30 metres” to support team collaboration. The presented studies yield the important insight, that different steps in the development cycle with different knowledge needs require different knowledge sharing channels.

Furthermore, distributed development introduces new kinds of knowledge needs. Besides knowledge about specific situations at the different sites (e.g. cultural habits, which are not under consideration here, since they are out of the scope of TEAM), it is primarily about task knowledge. In contrast to collocated development, where roles and responsibilities are rather clear and basic, task coordination may happen at lunch or at the water-cooler, distributed settings require mechanisms to create awareness of who is doing what.

Study	Research question	Method	Major results
The Object Oriented Team: Lessons for Virtual Teams from Global Software Development [100]	How do teams coordinate and conduct their work	Exploratory, interview, media richness theory; 4 GSD projects in three Indian software firms	Different coordination needs, different media; semantically rich media can decrease communication needs
Communication Needs, Practices, and Supporting Structures in Global Inter-Organisational Software Development Projects [98]	What communication needs and structures exist	Exploratory 32 interviews, 7 projects	Four important communication needs: problem solving, information and monitoring, relationship building, decision making/coordination
Architecture, Coordination, and Distance: Conway's Law and Beyond [90, 91]	What coordination problems exist in GSD	10 Interviews & Lucent study GB and DE	Coordination problems (lack of unplanned contact, Knowing whom to contact, • Cost to contact, Lack of trust) and best practices (Reduce need for coordination, Overcome barriers)
An Empirical Study of Speed and Communication in Globally-Distributed Software Development [92, 93]	Does the resolution process of modification requests, spanning multiple sites, take longer?	CVS and survey data at Lucent	Distributed work suffers from delays; use of informal communication because formal communication does not react quickly enough
Shared Mental Models,	How do shared mental models,	Multi-method-based test of	shared mental models and

Familiarity and Coordination: A Multi-Method Study of Distributed Software Teams [89]	work familiarity and geographic distance affect coordination?	three hypotheses: Three studies in one company (interviews, survey and archival studies)	familiarity with software and project reduces development time; effects are stronger for distributed teams
---	---	--	--

Table 5: Research in distributed/global software development

5.4 Open Source development

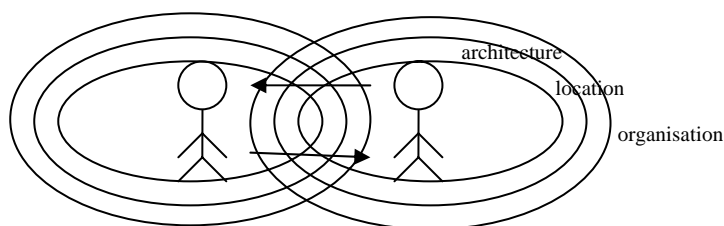


Figure 10: Relationships between developers in an Open Source team

The success of Open Source projects has attracted a large number of researchers to study this phenomenon.²⁰ Among different facets such as legal or economic issues, also work practices and developer interactions have become subjects of investigation. While one can argue that Open Source development is a special case of distributed Software Engineering, some additional issues do apply. Primarily, Open Source projects have more volatile project structure, partially lacking a clear organisational frame for all members, long-term participation and common vision of participants. In the following, we will describe some studies that address these special circumstances of Open Source development from a knowledge sharing point of view.

While participants in Open Source projects are often generally described as members of a core group, additional developers/bug fixers and simple developers, the actual structures might be different. Crowston and Howison [84] analysed the interactions from the bug tracking system of 120 projects on SourceForge²¹ to compute the communication centrality of the projects. They found out that projects differ largely in their structure, while larger projects tend to have more decentralised communication patterns.

Lanzara and Morner [95] did a detailed study to characterise the knowledge making process in Open Source projects. Therefore they analysed the Linux kernel and the Apache web server project. They found out that technology, rather than formal or informal organisation, plays a central role for knowledge making, which they describe as an ecological process where knowledge is never final but always in the making.

²⁰ <http://opensource.mit.edu/>

²¹ <http://sourceforge.net/>

A similar argument is made by Hemetsberger and Reinhardt [94], who observed the KDE project for four months. They conceptualise Open Source development infrastructure (e.g. code repository, mailing lists) as a transactive group memory, where the community reflects on the process of knowledge creation (e.g. by commit messages or e-mail discussions). Thus, Open Source projects enable a “re-experience” of past knowledge creation processes and learning from ongoing interactions to enter the project.

Thus, it turns out that the highly dynamic and “free” nature of Open Source projects poses new challenges for knowledge sharing. Basically, this boils down to the interrelated “maturity” of both developers and artefacts [81]. While the history of artefacts in controlled development settings is mostly important for auditing purposes, the described studies show, that artefact history can be important information for developers in a free setting. This is due to the fact that developers with different backgrounds join the effort at very different stages. Consequently, there is a need for new knowledge artefacts (e.g. FAQs) that are constantly maintained and serve as a reference for new co-workers.

Study	Research question	Method	Major results
The social structure of Free and Open Source software development [84]	Social structure/communication centrality of Open Source project	Analyze bug tracking data from 120 projects on SourceForge	Projects differ in communication centrality; larger projects have more decentralised communication patterns
The Knowledge Ecology of Open-Source Software Projects [95]	Characterise knowledge making in open-source projects	Explorative, Qualitative and quantitative analysis of Linux kernel and Apache web server	Open source is ecological system based on communication and artefacts
Sharing and Creating Knowledge in Open-Source Communities The case of KDE [94]	Knowledge sharing and creation process in an OS project (tools and forms and communication)	Four month observation, Grounded theory; community feedback regarding findings	Repositories form group memory that allow new participants to “re-experience” knowledge creation

Table 6: Research in Open Source work practices

5.5 Summary

This section has given an overview about empirical works related to Knowledge Management practices in Software Engineering. Although this is a rather new area, the presented studies show a broad number of different approaches. While especially the early works have a strong exploratory focus, employing qualitative methods such as observations or interviews, more recent works show a trend towards sophisticated theoretical frameworks and statistical analysis.

While all studies clearly deal with Knowledge Management topics, it is interesting to notice, that a majority does not draw from classical Knowledge Management conceptualisations. Instead, low-level activities such as coordination and communication are prevalent with few considerations about the possible impact on a higher knowledge process level.

However, basic knowledge needs can be extracted for the different phases as summarised in Table 7. While each sphere introduces some additional knowledge needs and makes knowledge sharing more difficult, there are two major insights from the studies on distributed and free development. The first one is that the suitability of channels for knowledge sharing depends on the actual step in the development cycle with its different knowledge needs. Secondly, Open Source development practices reveal, that both developers

and artefacts can be at a certain level of maturity, requiring a matching to ensure efficient knowledge sharing.

	Knowledge-related activities	Knowledge types	Knowledge media / Channel
Personal sphere	Search, Read, Talk, Document	Domain knowledge Meta-knowledge	Documents/Artefacts Communication (Face to face)
Team development	Work-package coordination Work practice coordination	Architectural knowledge Work practice knowledge	Communication (Face to face, meetings)
Distributed development	Process coordination	Task knowledge Cultural knowledge	Mail, instant-messaging Groupware
Open Source (“Free”) development	Contributing Bug-fixing	Historic/Evolutionary knowledge	Artefacts Mailing lists, Development tools

Table 7: Predominant/specific knowledge issues in software development spheres

6 CHALLENGES & FUTURE WORK

Implementing Knowledge Management in any organisation faces many challenges since, in order to start having return on investment, it requires considerable amount of effort and time. Implementing Knowledge Management in software organisations is even more difficult since the available time is less due to the fast pace of the Software Engineering business. The lack of time constitutes a direct threat against Knowledge Management. People often have no time to even search for knowledge resulting in not assigning the appropriate priority to a long-term investment such as Knowledge Management. This is mainly a cultural issue and as long as the management of the organisation does not allow the culture to change and does not allow employees to invest in managing their knowledge, then every Knowledge Management initiative is probably destined to fail.

Another challenge lies in the elusive nature of software itself. Software is not visible (in comparison to buildings in the civil engineering domain) and therefore it is often the case of being less reused than expected. Reinventing the wheel is a common phenomenon in Software Engineering since software developers are not provided with the capacity to discover previous, related work. Thus, software developers are less productive and the notion of reuse is something they are definitely not familiar with. The latter constitutes a hindrance to successful Knowledge Management initiatives in Software Engineering since reusing assets is a key aspect of Knowledge Management.

One of the inherent hindrances that Knowledge Management in Software Engineering has to face is the fact that most of the knowledge in Software Engineering is tacit and, most probably, will never become explicit. This is mainly the case due to the lack of time in making it explicit and the fact that software developers are reluctant in sharing their knowledge. Software developers feel possessive about their knowledge and are reluctant to share it by fear that they will become expendable [19]. The following are some reasons why employees would not share knowledge:

- The knowledge they have is why they are valuable to the organisation, why they are paid by the organisation and why they do not want to give it away.
- The thought that prevails in their minds is that they will be expendable as soon as their employers have captured all of the knowledge they need.

A way to address this problem could be to develop a knowledge sharing culture, as well as technology support for Knowledge Management. Creating a sharing culture is a long-term goal and presupposes that management creates trust amongst employees and between employees and management. People should be encouraged to share experience and help others and should be rewarded based on how much they share. There are numerous ways of rewarding employees and showing that the organisation appreciates employees who are willing to share their knowledge with others and are willing to search for and use knowledge created and documented by others [1].

TEAM will provide a new personalised, context-sensitive and decentralised system for sharing software development knowledge that can be seamlessly integrated in a software development environment. There are several challenges in realising such a knowledge sharing support. First, experience shows that users tend to be reluctant to provide the feedback about their experience, so that methods for capturing user's knowledge implicitly are required. Second, the experience should be represented in a form that allows flexible combination of information relevant for a particular problem. Third, the abstract organisation of knowledge and working context is required for establishing an efficient similarity measure needed for proactive knowledge delivery.

TEAM intends to address the aforementioned challenges and provide:

- An automatic elicitation of the knowledge through the concept of a context observer that is integrated in a software development environment (IDE).
- A distributed organisation of the knowledge (by using P2P infrastructures).
- An efficient search mechanism that enables finding similar but for the given context very relevant knowledge items.
- Proactive and personalised knowledge delivery depending on a user's working and personal context (provided in an unobtrusive manner).

7 REFERENCES

- [1] Davenport, T. H. and Prusak, L. (1998), *Working Knowledge: How Organizations Manage What They Know*, Harvard Business School Press, Boston, MA.
- [2] Birk, A., Surmann, D., and Althoff, K.-D. (1999), *Applications of Knowledge Acquisition in Experimental Software Engineering*, 11th European Workshop on Knowledge Acquisition, Modeling, and Management, pp. 67-84.
- [3] Bennis, W. and Biederman, P. W. (1998), *None of Us Is As Smart As All of Us*, IEEE Computer, Vol. 31, No. 3, pp. 116-117.
- [4] Lindvall, M. and Rus, I. (2000), *Process Diversity in Software Development Processes*, IEEE Software, Vol. 17, No. 4, pp. 14-71.
- [5] Basili, V. and Rombach, H. D. (1991), *Support for Comprehensive Reuse*, IEEE Software Engineering Journal, Vol. 22, No. 4, pp. 303-316.
- [6] Basili, V. R., Lindvall, M., and Costa, P. (2001), *Implementing the Experience Factory Concepts as a Set of Experience Bases*, Knowledge Systems Institute, 13th International Conference on Software Engineering & Knowledge Engineering, pp. 102-109.
- [7] Brössler, P. (1999), *Knowledge Management at a Software Engineering Company – An Experience Report*, Workshop on Learning Software Organizations, LSO'99, Kaiserslautern, Germany, pp. 163-170.
- [8] Henninger, S. (1997), *Case-Base Knowledge Management Tools for Software Development*, Automated Software Engineering, Vol. 4, pp. 319-340.
- [9] Tiwana, A. (2000), *The Knowledge Management Toolkit: Practical Techniques for Building a Knowledge Management System*, Prentice Hall PTR.
- [10] McGarry, F., et.al. (1994), *Software Process Improvement in the NASA Software Engineering Laboratory*, CMU/SEI-95-TR-22, Department of Computer Science, University of Maryland, College Park, MD 20742.
- [11] Basili, V. R., Caldiera, G., and Rombach, D. H. (1994), *The Experience Factory*, Encyclopedia of Software Engineering - 2 Volume Set, Wiley, pp. 469-476.
- [12] Ackerman, M. S. (1990), *Answer Garden: A Tool for Growing Organizational Memory*, Conference on Office Information Systems, COIS90, Cambridge, Mass.
- [13] Basili, V.; Costa, P.; Lindvall, M.; Mendonca, M.; Seaman, C.; Tesoriero, R.; Zelkowitz, M. (2001), *An experience management system for a software engineering research organization*, Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop.
- [14] Humphrey, Watts S. (2005), *PSP: A Self-Improvement Process for Software Engineers*, ISBN: 03213054931.
- [15] Humphrey, Watts S. (1999), *Introduction to the Team Software Process*, ISBN: 0-201-47719-X.
- [16] Humphrey, Watts S. (1999), *Pathways to Process Maturity: The Personal Software Process and Team Software Process* (Available at : <http://www.sei.cmu.edu/news-at-sei/features/1999/jun/Background.jun99.pdf>).

- [17] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995), Design Patterns Elements of Reusable Object-Oriented Software, Addison-Wesley.
- [18] Holz H. (2003), Process-Based Knowledge Management Support for Software Engineering, Doctoral Dissertation, University of Kaiserslautern, dissertation.de Online-Press.
- [19] Rus, I., Lindvall, M. & Sinha, S., S. (2001), Knowledge Management in Software Engineering A State-of-the-Art-Report. Fraunhofer Center for Experimental Software Engineering Maryland and the University of Maryland for Data and Analysis Center for Software, Department of Defence, USA.
- [20] Chung L., Nixon B.A., Yu E. & Mylopoulos J. (1999), Non-Functional Requirement in Software Engineering, Kluwe Academic, Boston.
- [21] Wikipedia (2006), Document Management Systems, http://en.wikipedia.org/wiki/Document_Management_Systems
- [22] NewHyperG AG (2006), Hyperwave IS/6, <http://www.hyperwave.com/e/products/is6>
- [23] Nguyen, T., N. & Munson, E. (2003), The Software Concordance: A New Software Document Management Environment, in Proceedings of the 21st International Conference on Computer Documentation (ACM SIGDOC), ACM Press.
- [24] Collard, M., Maletic, J., I. & Marcus, A. (2002), Supporting Document and Data Views of Source Code, in Proceedings of the 2nd ACM Symposium on Document Engineering (DocEng2002), McLean, VA, November 8-9, pp. 34-41.
- [25] Harzallah, M., Berio, G. & Vernadat, F. (2006), Analysis and Modeling of Individual Competencies: Toward Better Management of Human Resources, IEEE Transactions on Systems, Man, and Cybernetics, Part A 36(1), pp. 187-207.
- [26] URL: <http://www.hrhub.com/storefronts/skillscape.html>
- [27] URL: http://www.skillsoft.com/products/competency_management/skillview/default.asp
- [28] Sutcliffe, A., Chang, W., C. & Neville, R. (2003), Evolutionary Requirements Analysis, 11th IEEE International Requirements Engineering Conference (RE'03), p. 264.
- [29] Czuchry, A.J., Jr. & Harris, D.R. (1988), KBRA: A New Paradigm for Requirements Engineering, IEEE Expert: Intelligent Systems and Their Applications, vol. 03, no. 4, pp. 21-24, 26-32, 34-35.
- [30] Fouque, G. & Matwin, S. (1992), CAESAR: A System for Case Based Software Reuse, 7th Knowledge-Based Software Engineering Conference, KBSE, pp. 90-99.
- [31] Smith, T. & Setliff, D. (1992), Knowledge-Based Constraint-Driven Software Synthesis, 7th Knowledge-Based Software Engineering Conference, KBSE, pp. 18-27.
- [32] Fouque, G. & Vrain, C. (1992), Building a Tool for Software Code Analysis: A Machine Learning Approach, 4th International Conference on Advanced Information Systems Engineering, CAISE, pp. 278-289.
- [33] Terveen, L. G. (1995), Living Design Memory: Framework, Implementation, Lessons Learned, Human-Computer Interaction, Vol. 10, No. 1, pp. 1-37.
- [34] Holmes, R. & Murphy, G., C. (2005), Using structural context to recommend source code examples, in Proceedings of the 27th International Conference on Software Engineering (St. Louis, MO, USA, May 15 - 21, 2005). ICSE '05, pp. 117-125.

- [35] Ye, Y. & Fischer, G. (2002), Supporting reuse by delivering task-relevant and personalized information, in Proceedings of the 24th International Conference on Software Engineering, pp. 513–523.
- [36] Henninger, S. (1991), Retrieving software objects in an example-based programming environment, in Proceedings of the 14th International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 251–260.
- [37] Smith, D. (1991), KIDS: A Knowledge-Based Software Development System, Automating Software Design, MIT Press, pp. 483-514.
- [38] Kant, E. (1992), Knowledge-Based Support for Scientific Programming, 7th Knowledge-Based Software Engineering Conference, KBSE, pp. 2-4.
- [39] Michail, A. (2001), Code web: Data mining library reuse patterns, in Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society, pp. 827–828.
- [40] Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M. & Kusumoto, S. (2003), Component rank: Relative significance rank for software component search, in Proceedings of the 25th International Conference on Software Engineering, pp. 14–24.
- [41] Rosson, M., B. & Carroll, J., M. (1996), The reuse of uses in Smalltalk programming, ACM Transactions on Computer-Human Interaction, Vol.3, Issue 3, pp. 219–253.
- [42] Hill, R. & Rideout, J. (2004), Automatic method completion, in Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 228–235
- [43] Cubranic, D. & Murphy, G., C. (2003), Hipikat: Recommending pertinent software development artifacts, in Proceeding of the 25th International Conference on Software Engineering, pp. 408–418.
- [44] McCarey, F., Ó Cinnéide, M. & Kushmerick, N. (2005), RASCAL: A Recommender Agent for Agile Reuse, Artificial Intelligence Review.
- [45] Leitao L. (2003), Detection of Redundant Code using R^2D^2 , SCAM, Third IEEE International Workshop on Source Code Analysis and Manipulation, p. 183.
- [46] Weiser, M. (1979), Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, PhD thesis, University of Michigan, Ann Arbor, MI.
- [47] Xu, B., Qian, J., Zhang, X., Wu, Z. & Chen, L. (2005), A brief survey of program slicing, SIGSOFT Software Engineering Notes 30, 2, pp. 1-36.
- [48] Oman, P., W. (2005), Software vulnerability mitigation as a proper subset of software maintenance, Journal of Software Maintenance and Evolution: Research and Practice, Vol. 17, Issue 6, pp. 379-400.
- [49] O'Keefe, M. & O'Cinneide, M. (2006), Search-Based Software Maintenance, in Proceedings of the Conference on Software Maintenance and Reengineering, CSMR, IEEE Computer Society, Washington, DC, March 22 – 24, pp. 249-260.
- [50] Mens, K., Poll, B. & Gonzalez, S. (2003), Using Intentional Source-Code Views to Aid Software Maintenance, 19th IEEE International Conference on Software Maintenance (ICSM'03), p. 169.

- [51] Désilets, A., Paquet, S. & Vinson, N., G. (2005), Are wikis usable?, in Proceedings of the 2005 international Symposium on Wikis (San Diego, California, October 16 – 18), WikiSym '05, ACM Press, New York, NY, pp. 3-15.
- [52] Chau, T. & Maurer, F. (2005), A case study of wiki-based experience repository at a medium-sized software company, in Proceedings of the 3rd international Conference on Knowledge Capture (Banff, Alberta, Canada, October 02 - 05), K-CAP '05, ACM Press, New York, NY, pp. 185-186.
- [53] Cook, C. & Churcher, N. (2006), Constructing real-time collaborative software engineering tools using CAISE, an architecture for supporting tool development, in Proceedings of the 29th Australasian Computer Science Conference - Volume 48 (Hobart, Australia, January 16 - 19), V. Estivill-Castro & G. Dobbie, Eds. ACM International Conference Proceeding Series, vol. 171, Australian Computer Society, Darlinghurst, Australia, pp. 267-276.
- [54] Froehlich, J. & Dourish, P. (2004), Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams, ICSE'04, IEEE Computer Society, pp. 387-396.
- [55] Medynskiy, Y., Ducheneaut, N. & Farahat, A. (2006), Using hybrid networks for the analysis of online software development communities, in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Montréal, Québec, Canada, April 22 - 27), R. Grinter, T. Rodden, P. Aoki, E. Cutrell, R. Jeffries & G. Olson, Eds. CHI '06, ACM Press, New York, NY, pp. 513-516.
- [56] Draganidis, F. & Mentzas, G. (2006), Competency Based Management: A Review of Systems and Approaches, Information Management and Computer Security, Vol. 14 Issue 1, pp. 51-64.
- [57] Bruegge, B. & Dutoit, A. (2004), Object Oriented Software Engineering Using UML Patterns and Java, 2nd Edition, Prentice Hall.
- [58] Grinter, R. E., Herbsleb, J. D. & Perry, D. E. (1999), The geography of coordination: Dealing with distance in R&D work. ACM.
- [59] Herbsleb, J. D. & Mockus, A. (2003), An empirical study of speed and communication in globally distributed software development. IEEE Trans. on Soft. Eng., Vol. 29, Issue 6, pp. 481–494.
- [60] Jacobson, I., Booch, G. & Rumbaugh, J. (1999), The Unified Software Development Process, Addison-Wesley, Reading MA
- [61] Rumbaugh, J. , Blaha, M., Premerlani, W. ,Eddy, F. & Lorenzen, W. (1991), Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ
- [62] Booch, G., (1994), Object-Oriented Analysis and Design with Applications. 2nd ed. Benjamin/Cummings, Redwood City, CA
- [63] Jacobson, I., Christerson, M., Jonsson, P. & Overgaard, G. (1992), Object-Oriented Software Engineering- A Use Case Driven Approach, Addison Wesley, Reading, MA.
- [64] Rumbaugh, J. , I. Jacobson, I., Booch, G. (1998), The Unified Modelling Language Reference Manual, Addison-Wesley Object Technology Series.
- [65] OMG Ontology Working Group, Official Homepage, Sep. 2005. <http://www.omg.org/ontology/>

- [66] W3C Semantic Web Best Practices & Deployment, Working Group, Ontology Driven Architectures and Potential, Uses of the Semantic Web in Software Engineering, Sep. 2005. <http://www.w3.org/2001/sw/BestPractices/SE/ODA/>
- [67] Hesse, W. (2005), Ontologies in the Software Engineering process, EAI
- [68] Knublauch, H. (2004), Ontology-Driven Software Development in the Context of the Semantic Web: An Example Scenario with Protege/OWL, Stanford Medical Informatics, Stanford University, CA (Available at: <http://smi-web.stanford.edu/people/holger/publications/MDSW2004.pdf>)
- [69] Maalej, W. (2005), Domain Independent Generation and Management of User Queries in Semantic Web Applications, ‘Diplom’ Thesis, 2005 Technical University of Munich.
- [70] Henninger, S. & Ashokkumar, P. (2006), An Ontology-Based Metamodel for Software Patterns, In Proceedings, 18th International Conference on Software Engineering and Knowledge Engineering, San Francisco.
- [71] Perry, D. E., Staudenmayer, N. & Votta, L. G. (1994), People, organizations, and process improvement. IEEE Software, Vol.11, Issue 4, pp.36–45.
- [72] Kraut, R. E. & Streeter, L. A. (1995), Coordination in software, development. ACM, Vol.38, Issue 3.
- [73] Kunz, W. & Rittel, H. (1970), Issues as elements of information systems, Working Paper No. 131, Institut für Grundlagen der Planung, Universität Stuttgart, Germany 1970.
- [74] Regli, W. C., Hu, X., Atwood, M. & Sun W. (2000), A Survey of Design Rationale Systems: Approaches, Representation, Capture and Retrieval, Engineering with Computers, Vol.6, pp. 209-235.
- [75] Dutoit, A. H. & Paech, B. (2003), Eliciting and Maintaining Knowledge for Requirements Evolution, In: Managing Software Engineering Knowledge, Springer.
- [76] Unified Modeling Language: Superstructure, Object Management Group 2004 <http://www.omg.org/docs/formal/05-07-04.pdf>
- [77] Gruber, T. R. (1993), A translation approach to portable ontologies. Knowledge Acquisition, Vol.5, Issue 2, pp.199-220.
- [78] Dutoit, A.H., McCall, R., Mistrik, I., Paech, B. (2006), Rationale Management in Software Engineering: Concepts and Techniques, In: Rationale Management in Software Engineering, Springer.
- [79] MacLean A, Young RM, Bellotti VME, Moran T (1996) Questions, Options and Criteria. In: Moran TP, Carroll JM (eds.) Design Rationale, Concepts, Techniques and Use, Lawrence Erlbaum Associates, Mahwan, NJ, pp. 147-184
- [80] Lee J. (1991) Extending the Potts and Burns model for recording design rationale. In: Proceedings of the 13th International Conference on Software Engineering (ICSE’13) IEEE Computer Society Press, Los Alamitos, CA, pp. 114-125.
- [81] Happel, H.-J. and Schmidt, A. (2007), Knowledge maturing as a process model for describing software reuse, To appear in Proceedings of the 9th International Workshop on Learning Software Organizations (LSO 2007), Potsdam, Germany. March 2007.

- [82] Carmel, E. (1999), *Global Software Teams: Collaborating Across Borders and Time Zones*. Prentice Hall PTR.
- [83] Cain, Brendan G., Coplien, James O., Harrison, Neil B. (1996), *Social Patterns in Productive Software Development Organizations*. In: *Ann. Software Eng.* 2, pp. 259-286.
- [84] Crowston, K. & Howison, J. (2005), *The social structure of Free and Open Source software development*. *First Monday*, Volume 10, number 2 (February 2005) (Available at: http://firstmonday.org/issues/issue10_2/crowston/index.html).
- [85] Curtis, B., Krasner, H., Iscoe, N. (1988), *A field study of the software design process for large systems*. In: *Communications of ACM* 31 (1988), Nr. 11, pp. 1268-1287.
- [86] Chau, T., Maurer, F. (2005), *A case study of wiki-based experience repository at a medium-sized software company*, K-CAP 2005, pp.185-186.
- [87] Cusumano, Micheal A. (1997), *How Microsoft Makes Large Teams Work Like Small Teams*, *Sloan Management Review* 39, Fall, Nr. 1, pp. 9-20.
- [88] Devanbu, P., Balzer, B. Batory, D., Kiczales, G., Launchbury, J., Parnas, D., Tarr, P. (2003), *Modularity in the new millennium: a panel summary*, *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*. Piscataway, NJ : IEEE Computer Society, pp. 723-725.
- [89] Espinosa, J., Slaughter, S., Herbsleb, J. (2002), *Shared Mental Models, Familiarity and Coordination: A Multi-Method Study of Distributed Software Teams*, *23rd International Conference on Information Systems*, pp. 425 – 433, Barcelona.
- [90] Herbsleb, James D., Grinter, Rebecca E. (1999), *Splitting the Organization and Integrating the Code: Conway's Law Revisited*, *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, New York: Association for Computing Machinery, pp. 85-96.
- [91] Herbsleb, J. D., Grinter, R. E. (1999), *Architecture, Coordination, and Distance: Conway's Law and Beyond*, *IEEE Software*.
- [92] Herbsleb, J. D., Mockus, A. (2003), *Formulation and preliminary test of an empirical theory of coordination in software engineering*, *Proceedings of the 9th European software engineering conference (ECSE/FSE)*, ACM Press, pp. 138-137.
- [93] Herbsleb, J. D., Mockus, A. (2003), *An Empirical Study of Speed and Communication in Globally-Distributed Software Development*, *IEEE Transactions on Software Engineering* 29, June, No. 6, pp. 481-494
- [94] Hemetsberger, A. and Reinhardt, C. (2004), *Sharing and Creating Knowledge in Open-Source Communities The case of KDE*, *The Fifth European Conference on Organizational Knowledge, Learning, and Capabilities*.
- [95] Lanzara, G. F. and Morner, M. (2003), *The Knowledge Ecology of Open-Source Software Projects*, *19th EGOS Colloquium*, Copenhagen.
- [96] LaToza, T. D., Venolia, G., DeLine, R. (2006), *Maintaining mental models: a study of developer work habits*, *ICSE 2006*, pp. 492-501.
- [97] Kraut, R. E., Streeter, L. A (1995), *Coordination in Software Development*, *Communications of ACM* Vol. 38, No. 3, pp. 69-81.

- [98] Paasivaara, M. (2003), Communication Needs, Practices, and Supporting Structures in Global Inter-Organizational Software Development Projects, Proceedings of the International Workshop on Global Software Development at the 25th International Conference on Software Engineering, Portland, Oregon, pp. 59–63.
- [99] Parnas, David L. (1972), On the criteria to be used in decomposing systems into modules, Communications of ACM Vol. 15 , No. 12, pp. 1053-1058.
- [100] Perry, D. E., Staudenmayer, N., Votta, L. G. (1994), People, Organizations, and Process Improvement, IEEE Software, Vol.11, No. 4, pp. 36-45.
- [101] Ramesh, V. & Dennis, A. (2002), The Object Oriented Team: Lessons for Virtual Teams from Global Software Development, In Proceedings of the 35th Annual Hawaii international Conference on System Sciences (Hicss'02)-Volume 1 HICSS, IEEE Computer Society, Washington, DC.
- [102] Seaman, Carolyn B., Basili, Victor R. (1998), Communication and Organization: an Empirical Study of Discussion in Inspection Meetings, IEEE Transactions on Software Engineering, Vol. 24, Nr. 7, pp.559-572.
- [103] Singer, J., Lethbridge, T.C., Vinson, N. G., Anquetil, N. (1997), An examination of software engineering work practices, CASCON 1997, Vol. 21.
- [104] Tichy, W. F. (1998), Should Computer Scientists Experiment More?, Computer Publication Vol. 31, Issue:5, pp. 32-40.
- [105] Treinen J. J. & Miller-Frost, S. L. (2006), Following the sun: Case studies in global software development, IBM Systems Journal, Volume 45, Number 4.
- [106] Tichy, W. F., Lukowicz, P., Prechelt, L., Heinz, E. A. (1995), Experimental Evaluation in Computer Science: A Quantitative Study, Journal of Systems and Software Vol. 28, Issue: 1, pp. 9-18.
- [107] Ward, J. & Aurum, A. (2004), Knowledge Management in Software Engineering - Describing the Process, Proceedings of the 2004 Australian Software Engineering Conference (Aswec'04), ASWEC. IEEE Computer Society, Washington, DC.
- [108] Walz, D. B., Elam, J. J. & Curtis, B. (1993), Inside a software design team: knowledge acquisition, sharing, and integration, Communications of ACM Vol. 36, Issue: 10, pp. 63-77.