A Documentation-Centred Approach to Software Design, Development and Deployment

Brigit van Loggem(✉) and Gerrit C. van der Veer

Open University Netherland, Heerlen, The Netherlands
brigit.vanloggem@ou.nl, gerrit@acm.org

Abstract. In this paper, we argue how a documentation-centred approach to systems design and development could provide the different roles involved in this activity with a common ground. A large heterogeneous development team can be seen as a Community of Interest, consisting of individuals brought together from different Communities of Practice. Each group brings to the CoI not only their own skills and experience but also their own values, mental models, working practices, and communication styles. Re-shaping documentation into a boundary object offers a solution to the dual problems of (1) heterogeneous mental models within a software development team and (2) the user support role being peripheral to the team. Documentation that evolves dynamically, changing shape as the development process proceeds, can support communication both internally (between members of a software development team) and externally (between developers and end users).

Keywords: Documentation · Systems development · Communities of practice · Communities of interest · Boundary objects · Shared mental models

1 Introduction

In software design, documentation is an object of neglect. While recognized as one of the deliverables of a software engineering project, very little of it is created with any degree of enthusiasm. User documentation is routinely shrugged off as "there mainly to make up for bad interaction design" and "never read, anyhow" [1, 2]; and systems documentation is seen as a necessary evil that developers prefer to avoid.

Yet forms of communication other than documentation are engaged in without complaint. In this paper, we highlight the core characteristic of documentation as a form of communication. We propose a documentation-centred approach to organizing the work in software development teams. This approach allows for documentation to do what it is best at, which is supporting communication: internally (between members of a software development team) as well as externally (between developers and end users).

First, we discuss a number of issues related to large heterogeneous development teams, noting how these consist of individuals brought together from different backgrounds. In order to work together towards a common goal, they must reconcile their different views on the system that they are building: a process that is not without difficulty. We then look at how the creation of user documentation such as manuals and online Help fits in with the overall development efforts, noting that this process is not without difficulty either. In the second section of this paper, we investigate how the notion of "boundary objects" may be applied to begin solving both areas of difficulty at the same time, and conclude by mentioning some of the challenges involved in implementing such an approach.

2 Co-operation and Information Exchange in Large Software Development Teams

Many different disciplines are involved in the design and development of any but the most trivial of information systems. In the early stages of computing the hardware formed the limiting factor. Software development was carried out by one programmer, often himself the intended user of the software. Very rapidly, however, the cost of hardware decreased; and the new possibilities offered by faster processors and disk drives and larger memories equally rapidly led to larger and more complex programs being written—"any program will expand to fill available memory", as a jocular maxim of computer science known as the Fifth Law of Computer Programming will have it. Soon, software systems became too complex for one individual to write. Nowadays, almost no commercial software is written by one single programmer. Much software even takes many dozens of man-years to develop, in a process known as "software engineering". The development of computer software has become a collaborative activity for which new languages have been developed, new working methods, and new professional specializations [3].

A quick and by no means exhaustive inventory of a number of IT-related job sites on the Internet conducted on 19 July 2012 revealed that it is no longer sufficient to open a can of programmers to have a software system built. Software engineering calls for project teams to be formed which may include not only programmers but analysts, application administrators, application programmers, application specialists, business analysts, business architects, documentation analysts, enterprise-wide information specialists, HCI designers, information architects, internet engineers, IT consultants, multimedia architects, network designers, network engineers, operations analysts, product specialists, requirements analysts, software analysts, software architects, software engineers, software test specialists, solutions specialists, support analysts, system administrators, systems developers, systems analysts, systems engineers, technical authors, technical consultants, technical designers, technical support engineers, test engineers, testers, trainers, usability designers, usability engineers, web designers, web developers, web producers and many, many more. All these bring to the work their own skill set, which may be any combination of some 75–100 skills directly related to software development[1].

[1] See: the Skills Framework for the Information Age (www.sfia.org).

This list does inevitably contain synonyms; especially as there exists no generally-accepted taxonomy of IT-related professions. In an attempt to steer clear of the bewildering array of job titles, we have opted to highlight the widely divergent backgrounds, interests, and skills represented in software development teams by distinguishing the following roles, each responsible for a different aspect of the to-be-built system:

– Functional analysts (FA). These are responsible for eliciting requirements and defining a functional specification of the system.
– System architects (SA). Based on the functional specification delivered by the analysts, the architects are responsible for defining a technical specification of the system.
– Interaction designers (ID). These are responsible for the usability of the system.
– Software programmers (SP). Based on the technical specification delivered by the architects, the programmers are responsible for writing the code.
– User support (US). These are responsible for supporting the end users of the system after it has been built, through user manuals and Help systems.

This simple description of a software development team is, of course, a gross over-simplification. It does, however, have the virtue of allowing us to acknowledge and discuss fundamental differences within such teams, without losing ourselves in subtle detail that is not pertinent to the line of our argument.

2.1 Communities of Practice and Communities of Interest

The different roles within a software development team are set off from one another not just by having different responsibilities and different skill sets, but also by belonging to different "Communities of Practice". A Community of Practice or CoP is made up of "practitioners who work as a community in a certain domain doing similar work" [4]. Any particular CoP has its own standards, values, and ways of doing things. Members of a particular CoP join that CoP's professional organization; read that CoP's professional literature; and learn "on the job" what is "relevant" and what is not. FA sees the to-be-built system in terms of its alignment to business requirements. SA sees it as an intricate construction of interrelated components. ID is involved with the user interface of the system, while SP's interest is with the way the system consists of blocks of code. US, finally, is set apart even further by focusing on a mental construct, known as the "User Virtual Machine" or UVM; which is defined as "not only everything that a user can perceive or experience (as far as it has a meaning), but also aspects of internal structure and processes as far as the user should be aware of them" [5]. The visible part of the UVM is what these authors refer to as the "perceptual interface" and what is more commonly referred to as the user interface (created by ID); but the UVM as a whole is a much larger conceptual machine that presents itself to the end user.

These different views on the to-be-built system become entrenched over time. Within each CoP, sustained engagement and collaboration leads to boundaries, based on a shared history of learning which is set off against that of other CoPs [6]. Knowledge remains localized, embedded, and invested in practice, so that it is difficult to share with outsiders [7].

When members of multiple CoPs are joined together in a team, with a view to jointly realize a particular well-defined result, they are said to form a Community of Interest or CoI. This is defined as a group of people "from different backgrounds coming together to solve a particular (design) problem of common concern" [4]. A software development team is such a CoI. The members of the CoI that is a development team bring to the efforts their own ideas as to what the system is about and how it works. They have different mental models of the system.

2.2 Mental Models

Although "mental model" is a term traditionally reserved for the understanding that a user constructs of a software artefact during the process of applying it to real-world tasks over a period of time, lately it has been extended [e.g. 8, 9]. By removing the condition of application to a real-world task from the definition, a user becomes any human actor who interacts with a software system over a period of time. Interaction then includes the interaction involved in the construction of the system.

It is not our intention to provide a complete review of the mental models literature. For a wide-ranging and multi-disciplinary overview of mental model theories and their various applications, see [10]. An older seminal work is [11]. Finally, [12] provides a thorough overview of mental models theories. There is little consensus on what the term "mental models" means exactly [see also 13, p. 73, and 14, pp. 109–111]. We can, however, distil a common narrative leaning most heavily on [15] and [16].

According to theory a *mental model* is continuously being constructed in the mind during interaction with a complex system, during all stages of learning from the very beginning all the way to the highest proficiency. Like any model, it is a simplified abstraction that is used to predict behaviours of the referent (the *target system*). In order to predict what the target system will do under certain conditions, the user will mentally apply those conditions to the model, "run" it, and (still mentally) observe the outcome. The model is seen as viable if running it results in reliable predictions about the behaviour of the target system. In situations where the target system is man-made, we can identify on the one hand the *user mental models* held by end users and on the other the *internal models*, held by the system's makers. Information on how the communication between end user and system unfolds may contain a *conceptual model*, which is any model that is explicitly worked out by the User Support role to stimulate meaningful learning in those being instructed.

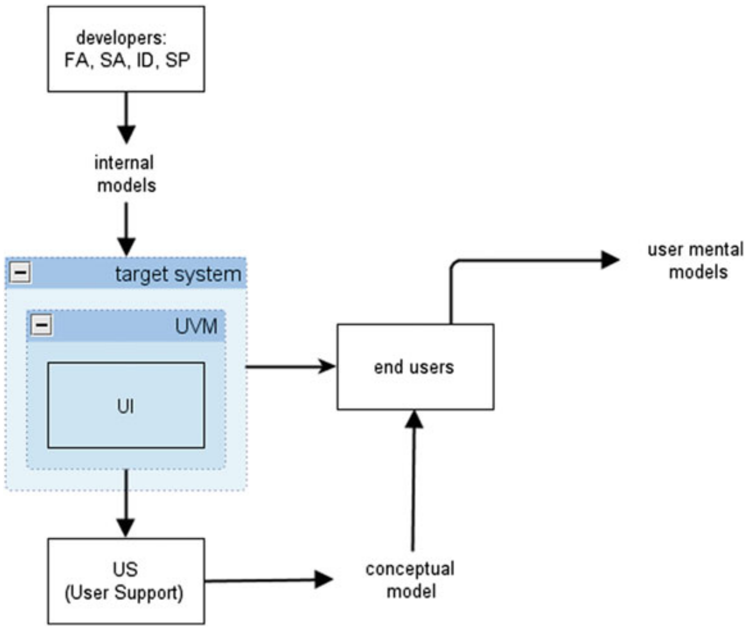

Fig. 1. Overview of mental models developed by different roles in the software engineering process.

As we have seen, the different roles within a software development team interact with different aspects of the system. Therefore, there is not just one internal model but rather a multitude. Rather than converging, team members' understanding of the system, the work, and other team members' expertise has been found to diverge over time. As time goes by, their mental models become increasingly dissimilar [9]. This is related to a decrease in interaction: the further into the project, the less team members engage in meetings and other such forms of communication.

2.3 Communication and Co-operation

Where many people work together towards a common goal, they need to communicate: if they don't, then achieving the goal will be very difficult if not impossible [3, 17] and the CoI will be unsuccessful. It is a sad fact of life that this does indeed happen. Complex software projects regularly fail to meet expectations.

Face-to-face communication is natural and often found to be the preferred channel for information exchange within a software development team. LaToza et al. found that programmers prefer face-to-face communication over every other method for obtaining an understanding of code written by somebody else [8]. However, face-to-face communication goes unrecorded and has spatial and temporal restrictions. Geographically distributed development teams are increasingly common and the system, once built, must be maintained and supported by others than those involved in the original development efforts [18]. For these reasons, most software engineering approaches call for extensive documentation of the development process. A potentially very large number of documents is created in the course of developing a software system [19].

Unfortunately, all this paperwork does not necessarily serve the desired communication. Programmers have been shown to conform to stereotype and go to great lengths to develop an understanding of the code; but they turn to its documentation only when everything else fails. Internal design documents are mostly read by newcomers to a team [8]. Furthermore, programmers have a strong sense of personal ownership of the code and hold enormous amounts of knowledge on the system in their heads. Their concept of "team" is limited to a very small number of direct colleagues, working on the same part of the code; and it is within these small teams only that achievements are documented. LaToza and his co-authors further found that for programmers the documentation serves not so much for information exchange as for information protection; for digging, as the authors call it, a "moat" around the work that has been done. Rather than describe the internal workings of a particular piece of coding, the documentation delineates the code by providing detail on its interface with other code.

To know which document to turn to in order to satisfy a need for information, an understanding is required of all that is available. The closer people are to each other in the team, the more they have the same understanding of what a particular document is good for; to such a degree that it has proven possible "to reconstruct an approximation of the development process based on statements solely about the documentation" [19]. As design documentation documents are read as well as written within the team, it follows that most of the internal documentation is well understood only by those whose work is closely related to that of the author. Communication within the CoI is severely hindered by the different representations that the different CoPs use for external cognition [4], and for actors to reconcile the different meanings is labour-intensive [20].

2.4 User Documentation

Thus far, we have discussed the documentation produced by the User Support (US) role only tangentially; and paid attention mostly to the documentation produced and used by the other roles in the development team. Where the latter's function is one of internal communication within the CoI, be it distributed or co-located and concurrent or over time, the former embodies communication with stakeholders outside the team: the end users of the system. For US documentation is an end rather than a means. This sets US off from the other roles in the CoI even more than the other roles are set off from one another (see Fig. 1). US's primary responsibility is to produce user manuals and Help systems. They describe the system after it has been completely built rather than during its construction; and indeed, frequently they are not even part of the development team but called in at a later stage, when the CoI is about to be or even has already been disbanded. US do not create that aspect of the system that they describe (the UVM) but identify the UVM as it emerges from the other roles' efforts. A software system is a self-contained "world" with its own objects (think of the Clipboard in many operating systems; of templates, style sheets and fields in a word processing environment; or of layers in an image editor). These software-specific objects, with their mutual dependencies and the rules governing their behaviour, are as much part of the UVM as is the interaction layer through which they are accessed. A user needs a thorough understanding of the UVM to gain complete mastery of a particular software tool, and apply it successfully to every task it can possibly be applied to. Such understanding is fostered through meta-communication in the shape of documentation or training [16]. As a "correct" user mental model is crucial to the end user's gainfully applying the system to real-world work, US strives to guide the formation of such a correct model; by explicitizing established misconceptions and subsequently eradicating them [21], or by presenting a conceptual model as depicted in Fig. 1 [22, 23].

In order to create a conceptual model for end users to learn from, US first need to develop their own correct mental model of the UVM. They do so by studying the internal documentation left behind by the development team and by holding formal or informal interviews with those of the development team who are still available. Then, they apply their knowledge of documentation and instruction to the scattered knowledge gleaned. This is a rather haphazard process, the result of which is often unsatisfactory. There is a strong need for US to be truly part of the CoI that creates the system. Only then will they be able to create user support materials that are genuinely helpful to end users.

3 Documentation as a Boundary Object

All design efforts, including the design of complex software systems, require the sharing of work artefacts [3]. Incremental creation of external representations is a strong mechanism for negotiating a shared understanding of the task at hand [4, 24]. In a seminal article [20], Susan Leigh Star presented the concept of boundary objects binding together heterogeneous groups of actors. Star's own words cannot be surpassed in describing the concept, as follows: "objects which inhabit several intersecting social worlds […] *and* satisfy the informational requirements of each of them. Boundary objects are plastic enough to adapt to local needs and the constraints of the several parties employing them, yet robust enough to maintain a common identity across sites. They are weakly structured in common use, and become strongly structured in individual-site use. These objects may be abstract or concrete. They have different meanings in different social worlds but their structure is common to more than one world to make them recognizable, a means of translation. The creation and management of boundary objects is a key process in developing and maintaining coherence across intersecting social worlds" [20, p. 393].

The current approach to documentation, in which every CoP within the CoI develops their own documentation separate from the others, is unsatisfactory; for a number of reasons:

- It further reinforces existing barriers (or "digs moats") between the different groups [8], as always one CoP is forced to discuss a document written in another CoP's language.
- It leads to information being lost, as translations have to be made between the different perspectives on the system.
- It is resource-intensive, as the different documents are all created from scratch.
- It is wasteful, as the separate documents have a limited life-span that is restricted to a particular development stage.
- It offers no guarantees for providing an exact description of what has been actually built at any given moment in time, as documenting and designing/developing are separate activities.
- It backfires, as people lose track of what has been documented; in which document a particular information item can be found; and where the different documents are stored [19].

In this paper, we propose a solution to the dual problems of (1) heterogeneous mental models within a software development team and (2) the user support role being peripheral to the team. The solution we propose is based on re-shaping documentation into a boundary object.

3.1 A Revised Role for Documentation

When carefully thought-out, one and the same set of documents could fulfil the roles of internal design documentation and user documentation. A semi-formal structure can be envisaged in which user requirements are laid down, after which the resulting documents are at every step further refined so that they become first the design specification and finally the user documentation.

A semi-formal description is one that combines the rigidity of a formalism with the flexibility of narrative. Within a pre-described framework, where building blocks are identified by (for example) fixed headings, a document's required content can be assembled at any given stage. Such a document is accessible to and can be a base for discussion between FA, SA, ID, SP, US and end users, who can then co-operate throughout the development cycle without loss and without spending any time or other resources in duplication; developing an ever-more detailed shared mental model over time.

Figure 2 gives a schematic overview of what such a revised approach could look like. Requirements are collected by the prospective end users together with FA; through whichever method is deemed appropriate. The resulting requirements are then recorded in a semi-formal document. This is agreed on by all those involved, and handed over to SA and ID for transformation into a functional description of the User Virtual Machine or UVM. Before SP starts working on its implementation, the UVM may be evaluated together with the users by means of any prototype-based method. Whichever form the prototype takes, however, the final, agreed-on version will become part of the documentation. SP then designs the implementation details and builds the tangible product. Finally, the documentation is worked up into an As Built description. Parts of this will now function as user documentation, while other parts will serve as systems documentation to be referred to for maintenance and updates. The documentation will now always exactly mirror the current state of the system. As a result, significantly higher quality should be achievable at significantly lower cost.

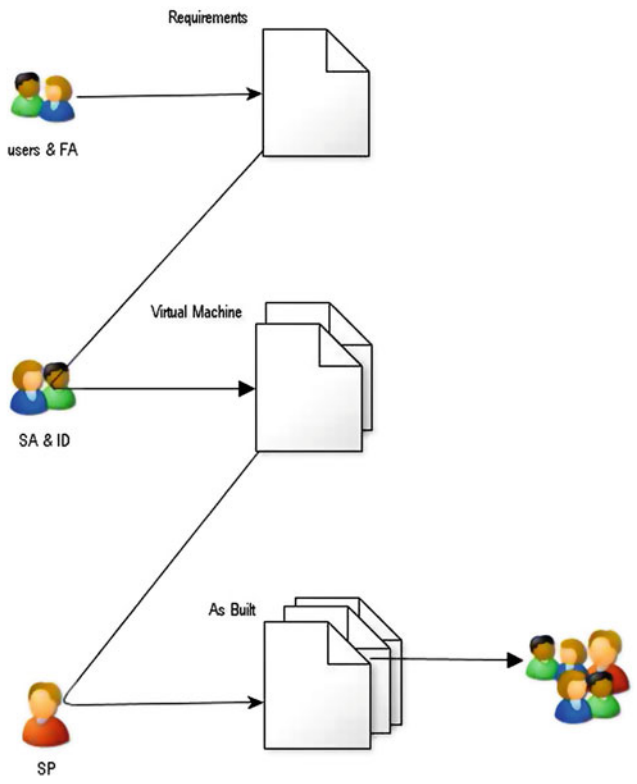

Fig. 2. One document, different stages, different uses.

In this approach, the documentation is at all times a boundary object in the meaning of the term defined by Star, inhabiting the worlds of all CoPs involved in the CoI. Being developed incrementally by all, it will satisfy the informational requirements of all parties, maintaining a common identity yet allowing for local detail. Bridging boundaries rather than digging moats, the ever-elusive quest for consensus is made redundant. "When participants in the intersecting worlds create representations together, their different commitments are resolved into representations – in the sense that a fuzzy image is resolved by a microscope. This resolution does not mean consensus." It is important to maximize not just the communication between worlds, but equally well their autonomy [20].

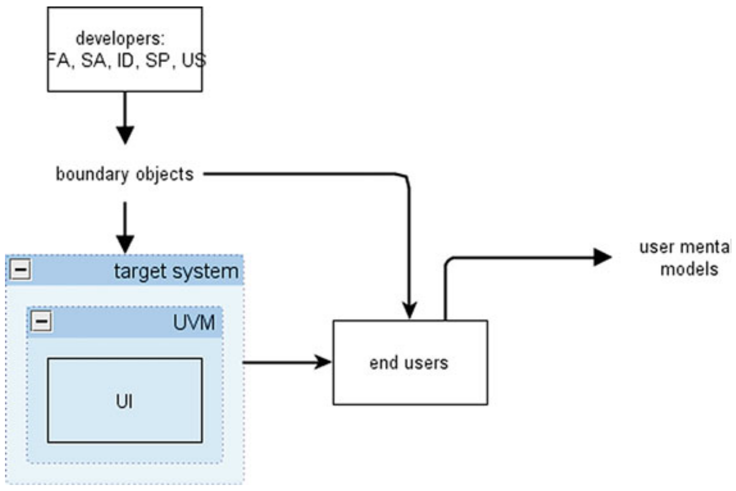

Fig. 3. Documentation as a boundary object.

3.2 A Revised Role for User Support

In the revised workflow described in the previous paragraph, mental models development is concentrated in the co-creation of documentation. Internal mental models as depicted in Fig. 1 are aligned as they develop. Moreover, the desired mental models of the intended end users are taken into account from the beginning. It follows then that there is no longer a need for User Support to separately develop their own mental models and then translate these into conceptual models to instruct end users (see Fig. 3). However, this does not mean that User Support has become superfluous. On the contrary, the particular expertise held by this role is indispensable now: not at the end of the development process, but rather throughout.

For boundary objects to function as such, standardization of methods is required [20]. As the process now hinges on one set of documents, any flaws in these will have repercussions at every subsequent stage. The semi-formal structure underlying the documentation must therefore be very carefully designed in its own right, to be tightly incorporated in the methodology that is applied. If this approach is to yield the desired benefits, a number of requirements must be met:

– The FA, SA, ID and SP roles must be supported in their endeavours to write in such a manner that the result is of value not only to themselves but to all other roles as well. This can be expected to require a major effort. Software developers are not known for either their willingness or their capacity to write and they will need to be motivated to do so. Then, they will need to be provided with instruction, guidelines and templates and most of all, they will need continuous monitoring and coaching.
– It must be clear who "owns" the documentation at any given stage. Transfer of ownership of the document marks the transition from one stage to the next.

- At every stage, the documentation must be able to incorporate additional new content such as requirements and features, systems logic and user interaction, and the nitty-gritty of coding solutions.
- Not only must the documentation provide room for all the content, it must also clearly prescribe the nature and scope of the content that is needed at every stage. A document set that is owned by different groups of people at different moments in time must be self-explanatory and self-directing: its contents cannot be left to chance or the personal preference of whoever happens to be working on a particular section.

Whereas initially the documentation consists of little more than a structure allowing for requirements and features to be recorded, the requirements analysis is concluded by creating "slots" for the design of the UVM by the SA role. This stage in turn is completed with not just production of the prototype, but also that of "slots" for the implementation design details that will be added by the SP role. Consolidation of all the information into an As Built deliverable that truthfully reflects that which has been delivered, finally, must be executed with a view to the final task of the documentation: that of supporting users on the one hand and providing a starting point for maintenance and further development on the other.

All these requirements mean that existing methodologies are no longer applicable and must be re-invented. Making software engineering documentation-centred can succeed only if the software development team includes a documentation specialist from the very beginning. User Support, with its core competence of documentation, is perfectly placed to truly make documentation into a boundary object.

4 Conclusion

Documentation has long been regarded as a necessary evil and treated accordingly. Internal design documents are discarded when the next stage of development begins. Development is often not documented beyond comments in the code, which tend to be few and far between. End user documentation is produced, if at all, at the very last moment by someone not involved in the design or development of the system. A recently proposed methodology for software engineering known as agile development calls for as little documentation as possible, and suggests internal communication is best carried out face-to-face. (Yet although agile developers on the shop floor may not be keen to produce documentation, they do feel that there should be more of it than there is [18].)

Documentation is recorded communication; and communication is what allows people to work together, creating something that one single person could not possibly achieve. Without communication, we cannot expect to build an interactive system that genuinely meets people's needs. The multitude of written documents produced between an interactive system's inception and the day on which it is last used, represent a missed opportunity for effective communication. All are written in their own language: some formal, some informal. Rather than facilitating the exchange of ideas and insights between different roles and across different stages of the system's life cycle, they scatter knowledge to such a degree as to make it effectively inaccessible.

The proof of the pudding is, of course, as always in the eating. Applying the approach described in this paper, in which one semi-formal, continuously evolving set of documents forms the focal point for everybody's contribution, will not be trivial. To test the approach in a real-life environment where a complex piece of software is developed to a real-life end, a large number of people need to believe in the concept and have enough faith in it to actually see it through. We believe that this will be an extremely rewarding and constructive exercise.

References

1. Mehlenbacher, B.: Documentation: not yet implemented, but coming soon. In: Jacko, J.A., Sears, A. (eds.) The HCI Handbook: Fundamentals, Evolving Technologies, and Emerging Applications, pp. 527–543. Lawrence Erlbaum, Mahwah, NJ (2003)
2. van Loggem, B.E.: User documentation: the Cinderella of information systems. In: Rocha, Á., Correia, A.M., Wilson, T., Stroetmann, K.A. (eds.) Advances in Information Systems and Technologies, vol. 206, pp. 167–177. Springer, Berlin (2013)
3. Barthelmess, P., Anderson, K.M.: A view of software development environments based on activity theory. Comput. Supp. Cooper. Work **11**, 13–37 (2002)
4. Arias, E.G., Fischer, G.: Boundary objects: their role in articulating the task at hand and making information relevant to it. In: International ICSC Symposium on Interactive and Collaborative Computing (ICC'2000) (2000)
5. van der Veer, G.C., van Vliet, H.: The human-computer interface is the system: a plea for a poor man's HCI component in software engineering curricula. In: Ramsey, D., Bourque, P., Dupuis, R. (eds.) 14th Conference on Software Engineering Education and Training, pp. 276–286. IEEE Computer Society (2001)
6. Wenger, E.: Communities of Practice: Learning, Meaning, and Identity. Cambridge University Press, Cambridge, England (1998)
7. Carlile, P.R.: A pragmatic view of knowledge and boundaries: boundary objects in new product development. Org. Sci. **13**, 442–455 (2002)
8. LaToza, T.D., Venolia, G., DeLine, R.: Maintaining mental models: a study of developer work habits. In: 28th International Conference on Software Engineering, pp. 492–501. ACM, Shanghai, China (2006)
9. Levesque, L.L., Wilson, J.M., Wholey, D.R.: Cognitive divergence and shared mental models in software development project teams. J. Org. Behav. **22**, 135–144 (2001)
10. Rogers, Y., Rutherford, A., Bibby, P.A. (eds.): Models in the Mind: Theory, Perspective and Application. Academic Press, London (1992)
11. Gentner, D., Stevens, A. (eds.): Mental Models. Erlbaum, Hillsdale, NJ (1983)
12. Schwamb, K.B.: Mental Models: A Survey (1990)
13. O'Malley, C., Draper, S.: Representation and interaction: Are mental models all in the mind? In: Rogers, Y., Rutherford, A., Bibby, P.A. (eds.) Models in the Mind: Theory, Perspective and Application, pp. 73–92. Academic Press, London (1992)
14. Payne, S.J.: On mental models and cognitive artefacts. In: Rogers, Y., Rutherford, A., Bibby, P.A. (eds.) Models in the Mind: Theory, Perspective and Application. Academic Press, London (1992)

15. Norman, D.A.: Some observations on mental models. In: Gentner, D., Stevens, A. (eds.) Mental Models, pp. 131–153. Erlbaum Press, Hillsdale, NJ (1983)
16. van der Veer, G.C., Tauber, M.J., Waern, Y., van Muylwijk, B.: On the interaction between system and user characteristics. Behav. Inf. Technol. **4**, 289–308 (1985)
17. Curtis, B., Krasner, H., Iscoe, N.: A field study of the software design process for large systems. Commun. ACM **31**, 1268–1287 (1988)
18. Stettina, C.J., Heijstek, W.: Necessary and neglected? An empirical study of internal documentation in agile software development teams. In: SIGDOC '11, pp. 159–166. ACM (2011)
19. de Boer, R.C., van Vliet, H.: Writing and reading software documentation: how the development process may affect understanding. In: 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering, pp. 40–47. IEEE Computer Society, Vancouver, Canada (2009)
20. Star, S.L., Griesemer, J.R.: Institutional ecology, translations and boundary objects: amateurs and professionals in Berkeley's museum of vertebrate zoology, 1907-39. Soc. Stud. Sci. **19**, 387–420 (1989)
21. Uzuntiryaki, E., Geban, Ö.: Effect of conceptual change approach accompanied with concept mapping on understanding of solution concepts. Instr. Sci. **33**, 311–339 (2005)
22. Mayer, R.E., Gallini, J.K.: When is an illustration worth ten thousand words? J. Educ. Psychol. **82**, 715–726 (1990)
23. Allbritton, D.W., McKoon, G., Gerrig, R.J.: Metaphor-based schemas and text representation: making connections through conceptual metaphors. J. Exp. Psychol. Learn. Mem. Cogn. **21**, 612–625 (1995)
24. Stettina, C.J., Heijstek, W., Fægri, T.E.: Documentation work in agile teams: the role of documentation formalism in achieving a sustainable practice. AGILE Conference (AGILE 2012), Dallas, TX (2012)