On the Process of Software Design: Sources of Complexity and Reasons for Muddling through

Morten Hertzum

Computer Science, Roskilde University
Roskilde, Denmark
mhz@ruc.dk

Abstract. Software design is a complex undertaking. This study delineates and analyses three major constituents of this complexity: the formative element entailed in articulating and reaching closure on a design, the progress imperative entailed in making estimates and tracking status, and the collaboration challenge entailed in learning within and across projects. Empirical data from two small to medium-size projects illustrate how practicing software designers struggle with the complexity induced by these constituents and suggest implications for user-centred design. These implications concern collaborative grounding, long-loop learning, and the need for a more managed design process while acknowledging that methods are not an alternative to the project knowledge created, negotiated, and refined by designers. Specifically, insufficient collaborative grounding will cause project knowledge to gradually disintegrate, but the activities required to avoid this may be costly in terms of scarce resources such as the time of key designers.

Keywords: User-centred design, Design process, Software development, Software-project complexity, Muddling through, Collaborative grounding.

1 Introduction

Software design is replete with projects that are cancelled, late, over budget, or result in systems with fewer features than originally specified [e.g., 5, 20]. Further, large numbers of systems are rejected by users or produce a merely marginal gain over former systems and work practices [e.g., 14, 28]. As an example, a recent national system for the Danish public administration was more than 100% late, more than 50% over budget, and reduced employee productivity by about 50% for several months after it was released. Six months after release an expert assessment concluded that considerable revisions of the system were immediately necessary, increasing the overspending to almost 100% compared to the original budget [12]. Troubled projects come about in spite of concerted efforts to the contrary, and they demonstrate the complexity of software design. Managing this complexity requires that its core constituents are well-understood.

This study analyses three constituents of software design and illustrates the analysis with empirical data from two projects. Each of the constituents is indicative of

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-540-92698-6_37

J. Gulliksen et al. (Eds.): EIS 2007, LNCS 4940, pp. 483–500, 2008.
© Springer-Verlag Berlin Heidelberg 2008

considerable complexity and – unless managed – entails serious risk to successful project completion. The analysed constituents of software design are:

- *The formative element*, which concerns articulating and reaching closure on a design
- *The progress imperative*, which concerns making estimates and tracking status
- *The collaboration challenge*, which concerns learning within and across projects

The formative element is at the core of human-computer interaction (HCI) and the two other constituents are crucial characteristics of the context in which practical HCI work takes place. Whereas the progress imperative has been acknowledged in much HCI work, for example the work on discount usability engineering [31], the implications of the collaboration challenge have not received nearly the same attention. This study aims to outline implications for user-centred design resulting from an analysis of the three constituents. For HCI researchers, the study intends to point out issues that may seem mundane but nevertheless hamper real-world projects, at least small to medium-size projects. For HCI practitioners, the study identifies some of the problems and tradeoffs they face in their work, and thereby offers an opportunity for reflection and pointers to means of alleviating some of the problems.

2 Empirical Data

To illustrate how practicing software designers approach the three software-design constituents that are analysed in this paper empirical data were collected from two software projects. The two projects are small to medium-sized and in this sense represent the majority of software projects [8, 17]. Neither of the organizations in which the projects took place follows a mandated design method but they have successfully completed a range of software projects.

The first project concerns a browser interface to a document-management system. Over a period of two decades the organization has developed, marketed, and continuously evolved a generic document-management system. The organization has 120 employees and a base of more than a hundred longstanding customers. Thousands of people use the document-management system on a daily basis. One high-level goal of this system is to provide professionals, as opposed to secretaries and document clerks, with easy access to organizational documents. In support of this goal it was decided to develop a browser interface to the system. The browser-interface project involved three designers and was successfully completed in seven months. The project was completed on time and within budget but this was partly achieved by reassessing and reducing the functionality of the browser interface halfway through the project.

The second project concerns a common user-interface platform developed by an organization that started by providing consultancy in hydraulic engineering but now increasingly develops and sells software instead of or along with the consultancy. The organization has 270 employees and has undertaken projects in more than a hundred countries. Over a period of three decades the organization has developed a number of hydraulic models and modelling tools as standalone software applications, but these applications generally have crude and inconsistent user interfaces and they must be ported individually to new operating systems. To mitigate these drawbacks a project

was established to provide a common user interface for the applications and handle their interaction with the operating system. The project, which involved 10-15 persons, took longer than planned and consumed more resources, but it was eventually completed.

For both projects two designers – the project manager and a programmer – were interviewed for a total of three hours. The obtained data are retrospective, though both projects were completed recently. In this sense the empirical studies are like post-project reviews. The interviews, which were audio recorded and subsequently transcribed, were loosely structured by a set of guiding questions. These questions concerned the major difficulties and information needs experienced during the project and the means in place to handle these information needs and communicate lessons learned. The interviewees' statements were compared and contrasted for purposes of validation. All interviewees were for the most part positive about their project but they also raised critical issues. Toward the end of the interviews, the interviewees were asked about their views on what had been the most significant risk factors in their project. This part of the interviews was based on a walkthrough of the 11-item list of top software-project risks identified by Schmidt et al. [36].

3 Three Constituents of Software Design

The project knowledge created, utilized, modified, embodied, shared, sought, and otherwise relied upon by designers must enable them to manage three complex and interrelated constituents of software design: the formative element, the progress imperative, and the collaboration challenge. Mapping these three constituents of software design to the lists of top software-project risks identified by Boehm [4] and Schmidt et al. [36] shows that the three constituents encompass the bulk of complexity that must be managed in software projects (Table 1). Of the 21 top risks on either of the two lists ten concern the formative element, five the progress imperative, and three the collaboration challenge. Only three risks, about limitations of technology, are not covered by the three constituents.

3.1 The Formative Element

The formative element is about articulating and reaching closure on a coherent design. After discussing this constituent of software design it is illustrated with data from the two empirical studies.

Articulating and Reaching Closure on a Design. The need for new systems can manifest itself in manifold ways, such as dissatisfaction with present ways of working, demands for new outputs, and knowledge of new technological options. This initial need provides only a vague or high-level specification of what is required from a new system and, consequently, software design involves a process of articulating the requirements toward the system in detail. The task-artefact cycle (Fig. 1 [9]) illustrates this cyclic and nontrivial process, in which designers respond to user requirements by building artefacts, which in turn present or deny possibilities to users. Users' understanding of their current artefacts is shaped by the tasks for which they are using the artefacts and, at the same time, their understanding of their tasks is shaped by the

Table 1. The coverage of the three constituents of software design in terms of the top software-project risks identified by Boehm [4] and Schmidt et al. [36]

Constituent	Boehm's top-10 [4]	Schmidt et al.'s top-11 [36]
The formative element: articulating and reaching closure on a design	▪ Continuing stream of requirements changes ▪ Developing the wrong functions and properties ▪ Developing the wrong user interface	▪ Changing scope/objectives ▪ Misunderstanding the requirements ▪ Lack of frozen requirements ▪ Lack of adequate user involvement ▪ Failure to gain user commitment ▪ Failure to manage end-user expectations ▪ Conflicts between user departments
The progress imperative: making estimates and tracking status	▪ Unrealistic schedules and budgets ▪ Gold-plating ▪ Shortfalls in externally furnished components ▪ Shortfalls in externally performed tasks	▪ Lack of top-management commitment to the project
The collaboration challenge: learning within and across projects	▪ Personnel shortfalls	▪ Insufficient/inappropriate staffing ▪ Lack of required knowledge/ skills in the project personnel
Other: limitations of technology	▪ Real-time performance shortfalls ▪ Straining computer-science capabilities	▪ Introduction of new technology

artefacts they currently use. Likewise, designers' understanding of the technological options is shaped by their knowledge of tasks that need to be performed and, at the same time, their understanding of users' tasks is shaped by the possibilities and restrictions of the artefacts they currently know of. Thus, people's familiarity with certain artefacts and certain tasks shape their understanding of what their tasks are and what technology has to offer, and this understanding, in turn, constitutes a perspective that points to certain technological options and makes people blind toward others [30]. This makes it inherently difficult for people to transcend their current way of perceiving things and envision how tasks, users, and technology should interact in constituting the future use situation.

The information needs inherent in the task-artefact cycle concern three areas of knowledge [27]: the users' present work, the technological options, and the new system. In a sense, the users' present work and the technological options are only of

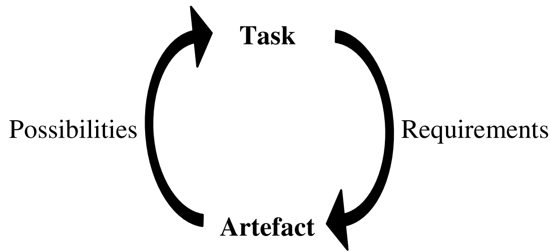

Fig. 1. Task-artefact cycle

interest because designers have no direct way of getting information about the new system and use situation. This is interesting from a project-knowledge point of view because it points out the massive indirectness of the information-seeking process in software design. Designers seek information about the users' present work, as opposed to their future work, and the technological options, as opposed to the future system, because they have no direct way of getting the information they really need. When designers are asked to design a new system they are, at the same time, prevented from getting crucial information about what properties this new system should have because people's familiarity with their present tasks and artefacts blocks their ability to envision radically new solutions. Further, software projects are frequently hampered by fluctuating and conflicting requirements because the learning process inherent in the task-artefact cycle continues throughout the projects and because the needs of different stakeholders may point toward different designs [4, 10, 36]. Apart from untangling these issues, which add to the difficulties of reaching convergence on a common project vision, requirements must not only be articulated they also need advocates. These advocates can be designers, users, or other people involved in a project. Eodice et al. [16] divided the requirements in a project they studied into those with and those without an advocate. They report that whereas virtually all the requirements with an advocate were eventually implemented not a single one of the requirements without an advocate were implemented.

Potts and Catledge [34] find that the process of reaching closure on the design of a new system is painfully slow and punctuated by several reorientations of direction. Lack of an agreed-upon understanding of what a system is to achieve complicates the development process because it leads to disagreements among designers as to the focus of the system and the best utilization of their resources. As a result, users may not be provided with any good system image [32] that presents the system facilities and their interrelationships in a clear and coherent manner. To provide insight about the use situation and thereby obtain a good match between user needs and system image prospective users must be actively involved in articulating and reaching closure on a design [e.g., 3, 18, 19]. At the same time requirements articulation is also a negotiation process in which designers need some level of control over the scope of projects to be able to balance their management of the contractual aspect of requirements specification against the facilitation of users in an open-ended search for requirements [23].

Browser-Interface Project. Two of the three designers involved in the browser-interface project had considerable knowledge of the users' work domain from previous

projects and could, thus, readily enter into discussions of requirements. The initial forum for these discussions was an annual two-day customer seminar hosted by the development organization to get feedback on released systems and discuss needs and ideas for new system facilities. For one of these seminars, which are attended by about 300 persons, a free-lance consultant made a prototype of a browser interface. Based on the feedback and discussions at the seminar it was decided to make the browser interface a top-priority project. This project was to provide platform-independent access to the document-management system without the need for installing additional software on users' computers. Further, the browser interface should be sufficiently undemanding to be usable without formal training, in contrast to the primary interface which requires a two-day course. While these high-level goals were clear from the outset a more detailed requirements document was never produced. Rather, the designers started coding early on and kept the evolving design partly in their heads and partly reflected in the code they produced. The intermediate outcomes of their work, in the form of system prototypes, were presented to and discussed with a group of user representatives with whom the designers met 4-5 times during the project. This led to the identification of a series of more detailed requirements, but the primary interface of the document-management system provided a default structure that significantly reduced the uncertainty and complexity involved in specifying the browser interface. The presence of the primary interface may, however, have rendered the designers and user representatives blind toward new possibilities and solutions. In continuation of this, one of the interviewees was concerned that the user representatives did not experience the prototypes in sufficient depth at the meetings and that actual use of the released browser interface might, therefore, give rise to many new requirements and change requests.

Common-Platform Project. At the overall level the common-platform project had a clear product vision from the very start, namely to provide a common, state-of-the-art graphical user interface for the individual hydraulic-engineering applications. Initially, the key person on the project was knowledgeable about both the hydraulic engineering that forms the basis for the applications and the user-interface programming that forms the basis for the common platform. This person has, however, left the organization and the remaining people on the project knew little about hydraulic engineering. Though the project members continually interacted with colleagues knowledgeable about hydraulic engineering this interaction was largely informal and the outcomes of these interactions remained in the heads of individual project members. No requirements specification was produced, discussed, iterated, and agreed upon, and apart from some code-level documentation the only up-to-date design documentation has been the project members' personal notes. The absence of systematic user involvement and requirements analysis provides strong candidate reasons for two of the three software-project risks identified by the interviewees as particularly relevant in relation to this project: failure to gain user commitment and failure to manage end-user expectations. The absence of design documentation such as an agreed-upon requirements specification also entailed that the project members were not supported in maintaining a shared understanding of the scope and objectives of the project. As a consequence there was no authoritative source in discussions about the functionality expected from different software modules and the project members repeatedly experienced difficulties in determining whether and when a module was complete.

Reasons for Observed Practices. Recommendations about how to articulate and reach closure on a design include principles such as "early focus on users and tasks" [18], techniques such as interpretation sessions [3], and artefacts such as requirements specifications. While such recommendations have been advocated for decades they are often not followed in practice [18, 34]. In the browser-interface and common-platform projects the main reasons for using proven design practices only sparingly were:

- *Believing high-level project goals are sufficient.* High-level goals like "providing platform-independent access to the document-management system" may provide a product vision but without complementary details the design is severely under-specified. Nevertheless, the designers in the two studied projects seemed to consider the high-level goals a satisfactory specification of their work in that they made no concerted effort to involve prospective users in producing a more detailed requirements specification.
- *Not knowing how to bring about more detailed requirements.* The designers seemed uncertain about how to get detailed requirements information from users and whether users would be able to provide such information. In the browser-interface project this uncertainty also included a fear of losing control over the process; that is, of eliciting requirements that went substantially beyond what they had the resources to deliver.
- *Focusing on the tasks they know best.* In a situation characterized by uncertainty and schedule pressure the designers concentrated on the tasks they knew how to do, primarily coding. This gave rise to a sense of progress though they were aware that important activities were being glossed over.

These reasons suggest that if given a structured process of clearly defined tasks for working systematically with requirements, designers will tend to follow this process [25]. But until such a process has become an established part of their repertoire many designers will likely muddle through the activities involved in articulating and reaching closure on a design.

3.2 The Progress Imperative

The progress imperative is about making estimates and tracking project status. After discussing this constituent of software design it is illustrated with data from the two empirical studies.

Making Estimates and Tracking Status. DeMarco [13] states that without estimates software projects cannot be managed. Estimation is a prerequisite for project planning which, in turn, provides for the coordination and management of design activities. Accurate estimates are, however, hard to make because the cost and time of developing both software modules and complete systems depend on multiple, interacting factors. Considerable experience is required to recognize the factors that warrant particular attention in a specified situation. Additional complicating factors include that individual differences in the productivity of experienced designers may be as large as 25:1 [15] and that requirement changes may necessitate rework. Inaccurate estimates of development cost and time impede the coordination of activities and allocation of resources both within and across projects. This may, ultimately, lead to

badly informed decisions about whether to continue or cancel projects. Consequently, the task of managing software projects involves that estimates are regularly checked against actual progress (Fig. 2). Estimates enforce plans by stipulating the amount of time and other resources allocated to a specified activity and must, at the same time, preserve realism by allocating enough time and resources to complete the activity. Conversely, status information enforces realism by accounting for how far the project has actually progressed and presupposes plans by assuming a shared understanding of what the outcome of specified activities should be.

Project-completion rates are low in software design [20, 36], and designers may thus be tempted to make optimistic estimates to avoid project cancellation, or they will simply direct their early efforts toward producing quick progress rather than spend their time on the planning that is necessary to make accurate estimates. DeMarco [13] finds that among software engineers an estimate is generally thought of as "the most optimistic prediction that has a non-zero probability of coming true". This leads to frequent underestimation. With appropriate training designers become better at estimating their work and the tendency to underestimate time and size is reduced, resulting in a more evenly balanced number of overestimates and underestimates [21]. These improvements are, however, inconsequential unless used, and it appears that estimates are often supplanted by performance goals, which are used to create incentives, or deadlines dictated by market pressures or other considerations external to the design effort. This implies that a consistent move toward more accurate estimates may require profound changes at the organizational and project levels in addition to an improvement in individual designers' ability to estimate their work [26].

Whenever a module is added or revised, ripple effects or previously undetected defects may emerge in other modules. Such changes to the status of modules are hard to predict and quantify ahead of time. In the absence of good estimation skills individual estimates may be made by increasing base estimates by a fixed percentage determined on the basis of accumulated experience. This is the approach taken by for example Microsoft, which adds 20-50% buffer time to base estimates [11]. Averaged over a number of activities such coarse-grained approaches may work well, but for individual activities designers will, at least occasionally, experience deviations that leave them idle for a period or block further progress on other activities. Organizations seem to work around these periods of waiting by assigning their designers to more than one project [33]. This, however, introduces additional dependencies that further complicate the plan-activity cycle (Fig. 2).

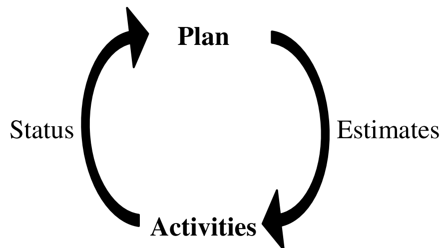

Fig. 2. Plan-activity cycle

Browser-Interface Project. The major means of managing the browser-interface project was two milestones. First, a working prototype should be ready for a meeting with the user representatives halfway through the project. Second, the system should be released at a fixed date. No tools or other formal means were in place to keep track of project status and support the designers in judging whether the project was on schedule. Rather, the designers relied on their personal sense of their progress and on extensive informal communication. Even formal meetings were few because the three designers were located close to each other – for part of the project they were in the same office. The designers' loose grip on status tracking was particularly evident in relation to testing. No established procedures for testing were in place and it remained, for example, largely untested whether system response times were acceptable and how platform-dependent they were. Similarly, the designers had no tools for managing their collaborative access to the source code, and there were incidents where they accidentally overwrote each other's files and thereby lost revisions. In the gradual process of setting the functionality of the browser interface the designers made explicit use of a multi-release strategy. That is, the top priority was to meet the project deadline whereas the functionality of the browser interface was considered malleable. This multi-release strategy exploited that the organization's document-management system already had an established position on the market and a base of customers that were as interested in being assured that the system grew in directions they considered relevant as in getting a specific piece of new functionality at a specific date.

Common-Platform Project. In the common-platform project progress toward satisfaction of requirements was not tracked systematically. Confidence in estimates gradually deteriorated and absence of shared agreement about the precise functionality of modules further eroded the basis for assessing module status. Contrary to this, an automatic mechanism was in place to track status at the code level and make updated versions of the code available to the designers. In total, the modules of the common-platform project comprise more than a million lines of code. The size of the code and the number of designers involved created a need for regularly establishing the code-level status of the modules and checking cross-module compatibility. This was achieved by a nightly build; that is, every night the latest version of each module was automatically compiled and linked with all the other modules. Whenever the nightly build succeeded the designers had a running version of their system. If a module contained errors that prevented its compilation or linking, it was automatically added to an intranet page listing the modules that failed the build, and an auto-generated email was sent to the designer responsible for the module. Thus, when the designers arrived at work in the morning they had access to a version of the code that included all designers' work up until yesterday evening and they had a complete list of the modules that failed the build. The nightly builds promoted a work practice in which people made an effort to check the correctness of their module before they went home. Further, some tests were run automatically every night with standard data sets and checks of system output against reference data. Finally, in-code comments were extracted from the code during the nightly build and a set of intranet pages generated. These web pages contained documentation of individual functions but rarely

covered interactions among functions or issues above the function level. Thus, while this documentation was regenerated every day it was insufficient as a means of making sense of the code. However, little design documentation exists apart from these web pages. The main reason for this is that the project group was under an unrelenting pressure to produce progress, and to be perceived as productive a designer had to be writing source code, not documentation. For similar reasons the status information resulting from the nightly builds was not accompanied by careful estimation and re-estimation of activities.

Reasons for Observed Practices. Reluctance or failure to make estimates and track status is widespread in software design. Common reasons for this are schedule pressure, fluid requirements, and limited experience with estimation [e.g., 4, 13, 25]. In the browser-interface and common-platform projects prominent reasons for the absence of systematic estimation and status assessment were:

- *Accurate estimates presuppose detailed requirements.* In the absence of clear requirements it is futile to attempt to estimate the time and resources required to complete a system or module. Rather, the designers in the browser-interface project reversed the process and used deadlines, which were stated more clearly than requirements, as a pragmatic basis for 'estimating' the functionality they would be able to deliver.
- *Not knowing how to handle estimates that are not met.* The designers in the common-platform project gradually lost confidence in estimation when they realized that they repeatedly failed to meet their estimates. Merely replacing old estimates with new made the whole effort seem pointless to them. Uncertainty and disagreements about the precise functionality of the modules further reduced their confidence in the estimates. Eventually, they largely abandoned estimation but kept tracking status.
- *Estimates are confronting for the individual designer.* Estimates create transparency with respect to whether the individual designer delivers on time or introduce delays that may have ripple effects on his or her colleagues' work. Thus, while estimates are central to the management of collaborative work, an immediate consequence for individual designers is increased exposure of delays and thereby a risk of being perceived as a less competent professional.

The nightly builds in the common-platform project illustrate that keeping track of project status at the code level and at the requirements level are distinct issues. Abstaining from working systematically with requirements means that decisions about requirements are made by individual designers and may subsequently be contested by other designers and by users. This provides a fragile basis for making progress and assessing project status.

3.3 The Collaboration Challenge

The collaboration challenge is about learning within and across projects. After discussing this constituent of software design it is illustrated with data from the two empirical studies.

Learning within and across Projects. In general, no single designer possesses all the required project knowledge in the necessary detail. Thus, to accommodate the customers' needs as well as needs arising from stakeholders such as marketing, service, maintenance, and quality control, software design becomes a collaborative effort. Another reason for developing software collaboratively is that many activities can then proceed in parallel and thereby both reduce the time from a decision is made to its consequences become apparent and shorten total development time. However, the distribution of software design onto multiple individuals creates a need for communication and coordination, which increases drastically with the size of the collaborating group [5]. Communication and coordination take place both within and across projects, corresponding to a short and a long learning loop (Fig. 3).

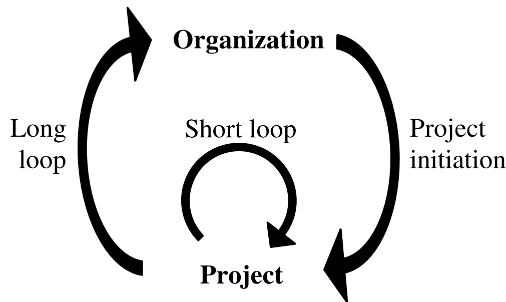

Fig. 3. Short and long learning loops

The project knowledge held by a group of designers is constantly evolving and in this sense learning is an integral part of their work practice [6]. This learning-in-working is local, aimed at competent performance, and woven into a collaborative practice. First, it is local in that it consists of gaining a coherent understanding of issues pertaining to the project at hand. These project issues are rich in contextual detail specific to the concrete situation, and these specific details are of paramount importance to the successful completion of projects. Second, it is aimed at competent performance because the ability to produce useful and usable systems in a well-managed way is much more salient to designers than production of generalized, explicit knowledge. According to Allen [1] this is the distinctive difference between engineering work and the work of scientists. Third, it is woven into a collaborative practice in that the different experiences and competencies contributed by different project participants provide learning opportunities beyond those available to people working individually. These learning opportunities enable designers to replace project activities involving prohibitive amounts of individual experimentation with close collaboration among people with relevant prior experiences.

Within projects written communication can be minimal if the designers meet often. Design methods often prescribe that a number of design artefacts are produced and kept up to date, but actual use of the methods tends to be more opportunistic [2, 22]. Design artefacts tend to be used at selected points in projects when designers perceive that the artefacts may have a direct impact on the progress of their project. During the in-between periods where the design artefacts are not contributing directly to the

designers' current activities the refinement and maintenance of the artefacts is likely to be postponed or downgraded in favour of activities that yield more immediate gains. Instead, designers carry most project information in their heads [34, 38]. This increases the reliance on oral communication and the centrality of the few people on a project who are able to reason and argue about how local changes affect the overall design. Over the course of a project these key people extend and refine their knowledge of the project by repeatedly debating alternatives, resolving disagreements, and incorporating redirections. Sharing this knowledge within the project group is an important but time-consuming process [3], and other project activities are likely to be competing for the key people's time, including activities that may appear more important because they break new ground and thereby yield pertinent project progress.

Across projects the experiences gained and solutions devised by designers may remain untapped by their colleagues because they are unaware of them or uncertain about their applicability outside their original context. The long loop represents this crucial but often unmanaged flow of experiences, solutions, and other knowledge from individual projects back to the organization for reuse in other projects. Zedtwitz [37] reports that 80% of projects are not reviewed after completion or cancellation to systematically and regularly make acquired project knowledge available for organizational learning. Further, in the design documentation made during projects designers are likely to make extensive use of condensed writing, which leaves most of the context unsaid because the documentation will be understood by its primary readers – usually other project members – as belonging to a certain ongoing activity. To make documents understandable to people who are not familiar with the context the condensed forms of writing must be elaborated, often to the exasperation of the primary readers who can see the elaboration as redundant [7]. Also, the pressure to produce project progress often precludes that designers spend time expanding their writings into documents understandable to unknown future readers [20]. Instead, most of the information that flows from project to project is carried by people, and oral communication and project staffing become key elements in the cross-project management of knowledge. This has spurred increasing interest in systems directed at locating knowledgeable colleagues – people-finding systems [e.g., 29].

Browser-Interface Project. The initial browser-interface prototype, and the analysis leading up to it, was made by a free-lance consultant who was not otherwise involved in the project. Thereby the three designers on the project missed the opportunity to learn from the consultant's experiences, apart from what they could deduce from the prototype. Instead, the three designers started largely afresh and relied on oral communication in keeping each other informed about their work. Written design documentation was sparse and played a negligible role. One of the interviewees estimated that a total of 20-25 pages of documentation were produced, all at the very end of the project. Apart from the small size of the project the interviewees emphasized three core success factors, all of which concerning the distribution of and easy access to project-relevant knowledge. First, the physical proximity of the three designers made it quick and easy to ask for help, and supported them in maintaining a mutual awareness of each other's current activities. Second, the three of them were responsible for the entire project. The absence of third parties enabled a way of working in which a shared understanding of the evolving design was constructed and maintained orally

through numerous conversations in their shared office. Third, the project was assigned one of the organization's most competent designers. The interviewed project manager stressed the importance of the few especially competent people and had made it a precondition for accepting to become the project manager that one of these core people was assigned to the project. Along with informal communication, staffing appeared to be the major way in which experience was transferred from project to project. In most cases staffing also determined the possibilities for reuse of software components because sparse documentation limited reuse to components the individual designers had themselves been involved in developing. The only occasion on which the browser-interface project has been evaluated and the lessons learned from it discussed was at an informal, project-internal meeting shortly after the project deadline.

Common-Platform Project. In the common-platform project the interviewees expressed a need for better ways of managing how far they had progressed toward completion. On the one hand, the project manager was not sufficiently good at defining and enforcing project milestones. On the other hand, the designers were not sufficiently good at communicating the actual status of their modules – many modules were "almost completed" for extended periods of time. The interviewees found that this boiled down to (1) frequent opacity or disagreements as to the functionality required from a module for it to be complete and (2) inadequate estimation skills. The first issue is a combination of communication breakdowns and imprecision in the analysis that turned overall project goals into specific requirements. This analysis was largely left to the individual designer, and no artefacts or stipulated procedures were in place to support the designers in communicating, arguing about, and reaching closure on the outcome of these analyses. A core element of the second issue is that writing source code was perceived as the primary activity whereas the time required for activities such as testing and documenting the code was generally underestimated. For the people appointed to system testing this activity was a secondary activity and their primary task consumed the majority of their time. Thus, testing was patchy and errors were encountered and corrected in a piecemeal fashion. The project did not include a post-project evaluation, and the organization has no cross-project forum for communicating lessons learned in one project to the rest of the organization. That is, the experiences gained in the project have not been the subject of collaborative discussion, apart from informal exchanges among designers. Thus, as an example, the nightly build and its associated mechanisms for supporting the development work were invented and instituted within the common-platform project by a single person, who has subsequently left the organization.

Reasons for Observed Practices. Projects are ubiquitous in software design, indicating that organized collaboration is biased toward the short loop whereas collaboration across projects tends to be informal [35, 37]. This is clearly illustrated by the browser-interface and common-platform projects. Apart from general cognitive and motivational factors [e.g., 24] reasons for having few artefacts and forums in place in support of the long loop include:

- *Short-term costs overshadow long-term gains.* Extra work is required to make project knowledge available to colleagues on other projects, and the reuse benefits of such work are hard to assess and more distant than the immediate tasks competing

for designers' time and attention. In small projects the extra work may be prohibitive and in highly dynamic settings reuse may seldom happen. However, the members of the browser-interface and common-platform projects felt that they ought to invest more in the long loop.

- *Project knowledge is context sensitive*. Designers interact repeatedly with their colleagues to get information, trusted opinion, and impetus for creative discourse. In these interactions, colleagues are not simply sources of information but actively involved in interpreting the applicability of their knowledge to the concrete situation. Conversely, designers are reluctant to engage in project post mortems and other activities that evolve around the context in which knowledge was gained because they are uncertain whether it will be applicable to future projects.
- *Not knowing how to make the long loop more effective*. A need for process support has been noted in relation to the two other constituents of software design but it is even more apparent in relation to the long loop. With the exception of documentation, the designers on the browser-interface and common-platform projects lacked knowledge of and experience with means of collaboratively managing the flow of knowledge across projects.

The collaboration challenge – especially the long loop – is the constituent of which the designers on the browser-interface and common-platform projects were least aware. At the same time, methods for managing the long loop appear to be less developed than for the short loop [24], though activities such as learning are crucially important to successful completion of software projects.

4 Implications for User-Centred Design

Based on the analysis of the three constituents of software-project complexity, this section aims to identify and discuss selected challenges to organizations' successful use and continued elaboration of practices for user-centred design.

4.1 Collaborative Grounding

In both empirical studies many of the troubles experienced by the designers concern collaborative grounding; that is, the active construction by actors of a shared understanding that assimilates and reflects available information. Project activities are rarely performed by the entire group of designers but typically by varying subgroups of the involved designers. Deliberate efforts of collaborative grounding are required to extend the knowledge acquired by a subgroup to the remaining designers on a project. The designers in the two empirical projects often under-recognized this need for collaborative grounding. Collaborative grounding is central to contextual design [3] and some participatory-design techniques [e.g., 19] but most techniques for user-centred design are biased toward information-seeking activities to the extent of largely bypassing collaborative grounding. For example, most usability evaluation methods focus on problem identification and largely evade the subsequent grounding of the evaluation results in the entire project group. This amounts to assuming that a project group is one unitary actor, rather than a network of actors that need to actively construct a shared understanding. The two studied projects vividly illustrate that the

designers struggled with collaborative grounding in relation to all three constituents of software design. Examples include that a shared understanding of module functionality was a long time in the making, that estimates were consequently inaccurate and difficult to interpret, and that no forums for long-loop learning were in place to prevent these issues from recurring in the next project.

4.2 Long-Loop Learning

Small project groups with around five members are widespread in software design, and many organizations actively opt for small project groups, for example by dividing development tasks onto multiple projects [8]. The browser-interface project is a case in point. In such small groups the communication and collaborative grounding necessary to cope with the short loop is manageable. Conversely, the common-platform project was staffed with 10-15 people, and this alone made it much more demanding to cope with the short loop. However, the size of a project group is also a means to shift the balance between the short loop and the long loop. A small project group needs frequent communication with project-external sources to exploit lessons learned in other projects. A larger project group will have access to more of these lessons by means of communication among project members and the long loop will, thereby, be partly subsumed in the short loop. Apart from project staffing, the organizations in both empirical studies relied on informal exchanges among designers as the principal means of exploiting experience from one project in other projects. Given the frequent recommendations of small projects [8, 11] and the ensuing reliance on an effective long loop it is noteworthy that methods for user-centred design focus almost exclusively on individual projects. Thus, methods as well as practitioners appear to devote most of their attention to the short loop and in so doing they render the long loop comparatively invisible. In both empirical projects the designers seemed to devote little time and attention to collaborative activities directed at improving their practices from one project to the next. Concrete guidance is needed on how to work effectively with the long loop in relation to user-centred design. Activities involving a more systematic pull of information, practices, and other resources into projects are probably more likely to become successful than activities aimed at pushing information and so forth from ongoing toward future projects.

4.3 Intimidation Barriers and Project Knowledge

The small to medium size of the projects and organizations in the two empirical studies could be an important factor in understanding their practices. The size may create an intimidation barrier toward software-process and long-loop initiatives that introduce (1) a new mindset promoting the longer-term effects of present practices rather than their more visible, immediate effects, (2) more systematic and regulated work processes, and (3) methods that are generally associated with large projects and organizations. The two empirical studies point toward a need for lightweight techniques and practices for managing the complexities inherent in the three constituents of software design. Discount usability engineering [31] suggests that unintimidating starting points and modest steps may be important to the adoption of such techniques and practices. However, practitioners also need to realize that as the systems they engage in

designing grow increasingly complex so does their need for techniques and practices that can match this complexity. A more managed process appears necessary. For user-centred design this seems to point toward further work on reaching closure on a design, integrating the task-artefact and plan-activity cycles, and communicating experiences across projects. Improved practices and a more managed process should, however, not be achieved by starting to consider methods an alternative to the project knowledge created by designers in response to the particularities of their current project.

5 Conclusion

Software design is a complex undertaking as evidenced by the frequency with which projects are cancelled, late, over budget, or resulting in marginal gains and systems disliked by users. Three major constituents of software-project complexity have been analysed in this study: the formative element, the progress imperative, and the collaboration challenge. Empirical data from two small to medium-size projects illustrate that practitioners struggle to manage these constituents. While each of the empirical studies is based on only two informants, the studies provide patent illustrations of a gap between the state of affairs in these software projects and the state of the art regarding software-process management. The designers in the two studied projects had few techniques and other means in place to support their work. Instead, they relied on an informal approach in which requirements, estimates, status information, and other design information were largely kept in the designers' heads and exchanged with close-by colleagues on an ad-hoc basis. The exceptions to this informal approach were carefully selected and mainly consisted of the nightly builds in the larger of the two projects and the annual customer seminar hosted by the organization in which the other project took place.

In many organizations, the principal means of coping with the long loop is project staffing. This reflects that project knowledge often unfolds around a few people with knowledge of relevant prior projects and the ability to take in the various pieces of information involved in a design, make out how they hang together, and articulate this clearly. A main challenge for user-centred design is to provide support for a more managed design process while avoiding that methods become seen as an alternative to project knowledge.

Acknowledgements. Johannes Knigge contributed to the empirical studies. Special thanks are due to the interviewees who agreed to participate in this study in spite of their busy schedules.

References

1. Allen, T.J.: Distinguishing engineers from scientists. In: Katz, R. (ed.) Managing Professionals in Innovative Organizations: A Collection of Readings, Ballinger, Cambridge, MA, pp. 3–18 (1988)
2. Bansler, J.P., Bødker, K.: A reappraisal of structured analysis: design in an organizational context. ACM Transactions on Information Systems 11(2), 165–193 (1993)
3. Beyer, H., Holtzblatt, K.: Contextual Design: Defining Customer-Centered Systems. Morgan Kaufmann, San Francisco (1998)

4. Boehm, B.W.: Software risk management: principles and practices. IEEE Software 8(1), 32–41 (1991)
5. Brooks, F.P.: The Mythical Man-Month: Essays on Software Engineering, Anniversary edn. Addison-Wesley, Reading (1995)
6. Brown, J.S., Duguid, P.: Organizational learning and communities-of-practice: toward a unified view of working, learning, and innovation. Organization Science 2(1), 40–57 (1991)
7. Brown, J.S., Duguid, P.: The social life of documents. First Monday 1, 1 (1996), http://firstmonday.org/issues/issue1/documents/index.html
8. Carmel, E., Bird, B.J.: Small is beautiful: a study of packaged software development teams. Journal of High Technology Management Research 8(1), 129–148 (1997)
9. Carroll, J.M., Kellogg, W.A., Rosson, M.B.: The task-artifact cycle. In: Carroll, J.M. (ed.) Designing Interaction: Psychology at the Human-Computer Interface, pp. 74–102. Cambridge University Press, Cambridge (1991)
10. Curtis, B., Krasner, H., Iscoe, N.: A field study of the software design process for large systems. Communications of the ACM 31(11), 1268–1287 (1988)
11. Cusumano, M.A., Selby, R.W.: How Microsoft builds software. Communications of the ACM 40(6), 53–61 (1997)
12. Danish Board of Technology: Erfaringer fra statslige IT-projekter – hvordan gør man det bedre? Report No. 10, Copenhagen, DK (2001)
13. DeMarco, T.: Controlling Software Projects: Management, Measurement and Estimation. Yourdon Press, Englewood Cliffs (1982)
14. Eason, K.: Information Technology and Organisational Change. Taylor & Francis, London (1988)
15. Egan, D.E.: Individual differences in human-computer interaction. In: Helander, M. (ed.) Handbook of Human-Computer Interaction, pp. 543–568. Elsevier, Amsterdam (1988)
16. Eodice, M.T., Fruchter, R., Leifer, L.J.: Towards a theory of engineering requirements definition. In: Lindemann, B., Meerkamm, V. (eds.) Proceedings of ICED 1999, vol. III, pp. 1541–1546. Technische Universität München, Garching, DE (1999)
17. Fayad, M.E., Laitinen, M., Ward, R.P.: Software engineering in the small. Communications of the ACM 43(3), 115–118 (2000)
18. Gould, J.D., Lewis, C.: Designing for usability: key principles and what designers think. Communications of the ACM 28(1), 300–311 (1985)
19. Greenbaum, J., Kyng, M. (eds.): Design at Work: Cooperative Design of Computer Systems. Erlbaum, Hillsdale (1991)
20. Grudin, J.: Evaluating opportunities for design capture. In: Moran, T.P., Carroll, J.M. (eds.) Design Rationale: Concepts, Techniques, and Use, pp. 453–470. Erlbaum, Mahwah (1996)
21. Hayes, W., Over, J.W.: The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers. Technical Report No. CMU/SEI-97-TR-001. Carnegie Mellon University, Pittsburgh, PA (1997)
22. Hertzum, M.: Making use of scenarios: a field study of conceptual design. International Journal of Human-Computer Studies 58(2), 215–239 (2003)
23. Hertzum, M.: Small-scale classification schemes: a field study of requirements engineering. Computer Supported Cooperative Work 13(1), 35–61 (2004)
24. Hinds, P.J., Pfeffer, J.: Why organizations don't know what they know: cognitive and motivational factors affecting the transfer of expertise. In: Ackerman, M.S., Pipek, V., Wulf, V. (eds.) Sharing Expertise: Beyond Knowledge Management, pp. 3–26. MIT Press, Cambridge, MA (2003)

25. Humphrey, W.S.: Why don't they practice what we preach? Annals of Software Engineering 6, 201–222 (1998)
26. Humphrey, W.S.: Three process perspectives: organizations, teams, and people. Annals of Software Engineering 14, 39–72 (2002)
27. Kensing, F., Munk-Madsen, A.: PD: structure in the toolbox. Communications of the ACM 36(6), 78–85 (1993)
28. Landauer, T.K.: The Trouble with Computers: Usefulness, Usability and Productivity. MIT Press, Cambridge, MA (1995)
29. Mockus, A., Herbsleb, J.D.: Expertise browser: a quantitative approach to identifying expertise. In: Proceedings of ICSE 2002, pp. 503–512. ACM Press, New York (2002)
30. Naur, P.: The place of programming in a world of problems, tools, and people. In: Kalenich, W. (ed.) Proceedings of IFIP Congress 65, Spartan Books, Washington, DC, pp. 195–199 (1965)
31. Nielsen, J.: Usability Engineering. Academic Press, San Diego (1993)
32. Norman, D.A.: Cognitive engineering. In: Norman, D.A., Draper, S.W. (eds.) User Centered System Design: New Perspectives on Human-Computer Interaction, pp. 31–61. Erlbaum, Hillsdale (1986)
33. Perry, D.E., Staudenmayer, N.A., Votta, L.G.: People, organizations, and process improvement. IEEE Software 11(4), 36–45 (1994)
34. Potts, C., Catledge, L.: Collaborative conceptual design: a large software project case study. Computer Supported Cooperative Work 5(4), 415–445 (1996)
35. Schindler, M., Eppler, M.J.: Harvesting project knowledge: a review of project learning methods and success factors. International Journal of Project Management 21(3), 219–228 (2003)
36. Schmidt, R., Lyytinen, K., Keil, M., Cule, P.: Identifying software project risks: an international Delphi study. Journal of Management Information Systems 17(4), 5–36 (2001)
37. von Zedtwitz, M.: Organizational learning through post-project reviews in R&D. R&D Management 32(3), 255–268 (2002)
38. Walz, D.B., Elam, J.J., Curtis, B.: Inside a software design team: knowledge acquisition, sharing, and integration. Communications of the ACM 36(10), 63–77 (1993)

Questions

Jan Gulliksen:
Question: This kind of work usually focuses on projects that have failed. Did you try to find successful projects and see how they work? Or find out whether changing practices would make projects more successful?

Answer: We didn't select our projects for success or failure. Others have looked at success. Also looking at projects that have used user-centred methods will tell us something more.

Annelise Mark Pejtersen:
Question: Can you make such a sharp distinction between successful and unsuccessful projects?

Answer: I agree. If you ask different people they will also have different views about the project. Some people focus on process, and others on product.