Software Design Processes: A Note on Micro-Sociological Perspectives

Alexandros-Andreas Kyrtsis

Abstract

Studies on software design and software development, departing from theoretical perspectives based on current work in micro-sociological perspectives, are almost non-existent. The reasons for this can be traced back to the empirical, theoretical and methodological underpinnings of the dominant approaches in the social studies of information technologies, as well as to problems inherent to the subject matter. Ways out of theoretical and methodological impasses can be found through a better understanding of the characteristics of processes of the social organisation of arenas of software design, as well as through a closer examination of the value of certain aspects of recent micro-sociological theory. The main aspect stressed here is the importance of the interplay between memory and forgetfulness in micro-processes of the social constitution of software designing communities of practice, which also put relationships between actants and objects (tools, methodology manuals, inscription devices, and IT products) through a prism enabling the study of the emotional underlay of fluid but directed formations of contingent spaces of software design.

Understanding software design processes

How important are creative design practices for the world of software development and how do they fit within processes of developing and delivering products and services by IT companies or organisational IT divisions? Astonishingly enough, researchers in the social studies of information and communication technologies know very little about this. Micro-studies on how software designers really work and produce informational goods and artefacts, with immense impact on our lives, are almost non-existent. The few we have are laden with biases blurring subtle information and obstructing fruitful interpretations. The emphasis of social science micro-studies of software development and design processes is laid primarily on how IT specialists discuss constraints originating in

existing or prospective contexts of use (Crabtree & Rodden 2002; Crabtree 2004). It is true that requirements capturing and subsequent requirements engineering (especially accountable software engineering) by addressing what are deemed to be the needs of users, have shifted the focus towards the social life of informational goods and artefacts. But such efforts, as detailed workplace studies show (Luff, Hindmarsh & Heath 2000), rarely reach the point of going beyond studying perceptions or reactions of technologists encountering user environments. Also, research in the social shaping of information technologies, although in a slightly different sense, portrays processes of design and development as results of the internalisation of societal, economic and political influences and pressures.[1] This orientation is very often combined with an interest in the outputs of technological communities of practice, in other words in the impact of information and communication technology products and services upon a society of users. This has been the main line of the social studies of information and communication technologies and has led, as everyone knows, to the production of a vast literature in this field.

The emphasis on the user orientation of designers and developers as the basis of the critique of technocracy, however legitimate and productive it may be, is one of the reasons why our vision cannot effectively intrude upon software designing and software developing communities of practice. This is mainly due to 'design fallacy' which comes from the idea that design, if carried out as an inductive process of accumulating information about user and system requirements, can best meet eventual needs (Stewart & Williams 2005). The critical point here is that, however interactive design processes may become by involving enabled and empowered users at every step, the input stemming from the designer's imagination, intentionality and translations of data perceived dominate the cognitive and emotional basis of the entire creative design process. Studying the designer as situated in the world of the user, as Terry Winograd (1996, xvii) pleads, does not solve the problem of understanding the software engineer's, designer's or programmer's intentions of generating the operational and informational architecture of the space in which the user lives and works. As Robert Anderson (1994) has shown, this also applies to ethno-methodologically oriented studies which are justified by the insight that in many

cases the user's knowledge of the working context can be as vital as the designer's technical knowledge for the eventual success (or failure) of systems. A similar problem appears in the case of approaches drawing our attention to participatory design. Proponents of participatory design seek to illustrate design activities as the result of the reconciliation between user's and designer's perspectives, however they may differ (Floyd 2002, 27; Kaptelinin 2002, 60). Contrary to this view, the resulting user practices and enacted artefactual architectures in most cases emerge from the acceptance and 'domestication' of plans, which whenever made apparent reveal the limits of both social constructivism and of the supposedly even relationship between the designer and the user (Kallinikos 2004).

Methodologies of user oriented practices are still methodologies, deriving from decontextualised and abstract ideas. If these are taken for the standards of how we should view designer and development communities, analytic vision is blocked. Similar problems appear in reports on how software designers and developers work, set out by software specialists (Dittrich et al. 2002) or by managers (Cusumano 2004; Cusumano & Selby 1998). The theoretical underpinnings of these reports, in spite of any sociological and anthropological references, are rather technical or managerial, and this makes them digress from social scientific rigour. Orientation towards normative technical or managerial scenarios of action is the usual line along which researchers coming from software engineering try to fill the gap by initiating detailed studies of software development processes using methods and techniques of both quantitative and qualitative social research (Boehm et al. 2005; Kemerer & Slaughter 1997; Lethbridge et al. 2005; Singer at al. 1997). The aim of these analyses is to understand developers' behaviour in order to adjust it according to best practice standards. Understanding the human and emotional underlay of software design and development processes, with the explicit or implicit purpose of adjusting or even eliminating it, is the task of any technologist, not only that of the technocrat. Viewing technologies as settings in which humans can be disciplined is the dominant stance. What makes the difference is the approach to disciplining. Technological reductionism is not the only alternative. Some technological plans draw on the design of configurations of social role-sets. The micro-sociological sensitivity of

many of the studies carried out by IT specialists is much more related to an interconnection between, on the one hand, the imposition and acceptance of methodologies, and on the other systems of role distribution among practitioners who, willingly or not, accept these methodologies. Besides any more or less subtle technocratic ideologies, these studies are one-sided because they divert attention from the actants who try to weave disciplinary plots. The point is that technologies do not solely emerge from the symbolic orientations and the emotions of the ones who are disciplined, but also from the symbolic orientations and the emotions of the ones who try to discipline others in the name of technological rationality. The technological intention is not reflexively and critically discussed in these approaches and this distorts the whole of the analysis. In most cases methodologies of design as well as of requirements capturing, which structure deliberation culminating in shaping especially complex formal systems, can be a significant source of discourses of user communities delivering requirements: users' discourses are a direct or indirect reflection of designers' and engineers' discourses. In this sense many texts related to software and requirements engineering, although they can have a micro-sociological touch, seek to create configurations of social roles as part of working methodologies addressed to groups of IT specialists, or as part of technical prescriptions addressed to users who are supposed to have a greater ability for adopting technologies if they follow a certain role-play or script. Methodologists seek to create social roles, but also social role distributions appear as the basis of methodologies, as is the case with agent-based approaches to software development (Jennings 2001). What makes agent-based software engineering interesting is that it departs from role-sets which can be regarded as structural features of domains of social and organisational action corresponding to sets of computer applications. Versions of software engineering into which these kinds of translation evolve, are object-oriented software engineering or pattern language oriented software design.[2] In sum, the formal construction of techniques of role-set engineering, deriving from methodological prescriptions and assessment metrics, even if they rely on concepts borrowed from micro-sociological approaches and research in social psychology, is exactly the opposite of what we need. These analytic perspectives adopted by technologists belong to the technological practices

the social meaning of which will remain unexplored as long as our mindsets and the lack of appropriate field research make it difficult for any attempt to know more about software designers and software developers.

The dominant role of technologists in the study of software design and development processes is of course due to the fact that very few sociologists or anthropologists have the necessary technical knowledge. But even for the ones who belong to this minority, investing in time-consuming participant observation or narrative interviews seems not to be necessarily a sensible option. Besides, there are difficulties inherent to the subject matter: the propositional part of discourses in such communities is extensively bound to informational tools, inscription devices and congenial diagrammatic reasoning.[3] Appropriate skills of the observer often do not suffice; only practical involvement can, under certain circumstances, enable a thick description of the pertinent practices—not an easy task to cope with. Such lack of knowledge and skills is a secondary handicap, however, since most people have not even the feeling which might orientate them towards posing the relevant research questions. And practitioners, who might be both motivated and skilled, are certainly not preferable recruits for this job, as they almost inescapably lack the ability of observation from a critical distance. Missing sociological, anthropological or micro-historical entry points to the world of software people, which thus remains unknown to outsiders, are then combined with standards of technological relevance intensifying the narrowing of vision.

Another handicap of studying fields of software design and development arises from difficulties in demarcating the domain of inquiry. The discussion tends to run without realising that not all kinds of specialists in all kinds of communities of software people work under the same emotional and epistemic regimes. The emergence of the software designer, as a dominant role among others in multidimensional role-sets, is a consequence of the shift of software towards the symbolic and the cultural. This shift can also be observed in the role of the so-called system analysts, even sometimes in the roles of the so-called presales specialists. But these roles, contrary to other software development roles, are too much adapted to short-term problem solving and customised or versioned partial IT solutions.[4] User-orientation and solution design among analysts and

presales staff is then bound too much to sales strategies and probably also to asymmetric information between developers, vendors and users. Further, there is a crucial difference concerning the divergence between communities of computer scientists and programmers on the one hand, and software designers on the other. Or better, between communities where the one category dominates the other. Among the various categories of specialists there is one which tends to place itself between the two. These are the software engineers, who differentiate themselves from computer scientists and programmers by giving emphasis to what they call requirements engineering, but who are also different from software designers by putting emphasis on formalised methods of translating the social into the technical. Software engineers, despite any nuances, are always on the side of computer scientists and programmers, whereas software designers, a species appearing in the later stages of the evolution of informatics and gradually becoming dominant especially in innovative IT companies, despite any problems stemming from the design fallacy, are always very much preoccupied with understanding user contexts and users (Cusumano & Selby 1998, 79). Computer scientists and programmers, but also main-stream software engineers, are closer to the character of experts, whereas software designers are more like reflective practitioners.[5] Besides, computer scientists and programmers are oriented more towards auto-operative processes, whereas designers are more oriented towards technologically enabled human action in context (Oberquelle 2002, 398).

What also differentiates the various categories of computer specialists is the degree to which parting software from program is a priority with implications for their practical styles. Whereas programs can be reduced to algorithms, software is in addition associated with documentation and configuration of pieces of information needed to create the necessary knowledge which triggers modes of operation. In other words, whereas programs are sets of transformations of data into information, software extends and articulates the functionalities through transformations of pieces of information into knowledge and knowledge-bound practices.[6] Many software engineers (King 2005; Sommerville 2001, 5) make this distinction but believe in the possibility of eliminating, by purely scientific means, the tension between the first and the latter part of these processes

(MacKenzie 2001). This tension creates dividing lines between software and program, as well as between software engineering and software design—the latter being an activity oriented towards genuine attempts at understanding user contexts and users (and thus design configuration of activities and system functionalities) by trying to translate the technical into the human aspects and not vice versa. The social roles of translators of the technical into the human tend to become dominant in many contexts of software and information systems development. Translating the technical into the human can nowadays be very often the main preoccupation of designers of system functionalities, in spite of not being always widely accepted among informaticians. The culture of informatics historically stems more from the mentalities of programmers who are interested in specifying automatic computations or algorithms which are taken for the basis of solving well defined formal, as well as real world problems (Floyd 2002, 13–14). Programming relies on a clear separation of the human and computer elements, putting emphasis on functional and technical requirements and operational reliability. The problem we are facing then, because of the primacy of the formal and the technological element, is that this separation of programming language and the language of human relations implies an imposition of the former upon the latter, resulting in forms of decontextualisation or fragmentation of the social world in working contexts by equating social processes with program functionalities.

Contrary to computer scientists and programmers, software designers become deliberately involved in the elaboration of this contradiction between contextualisation and decontextualisation. The more software design and development requires metaphors and tensions of the contextualised world, the more these metaphors and tensions are transferred into the micro-fields of social interaction among software designers, developers and programmers. Despite the differences in both practical and discursive power, the subsequent co-evolutionary processes can enhance the ties between the dynamics of several software-producing communities of practice and the communities of users and operators of systems and remote devices. Thus general rules and methodologies acquire much more diverse and ambiguous uses in such enacted spaces of action, than is insinuated in the textbooks and handbooks.[7] Rules are construed and discursively

repackaged in a way that carries in it the dual intention to make utterances conventional both in terms of the culture of the guild of software specialists and in terms of the culture of the particular and socially differentiated communities of practice. Embedding computer artefacts in social contexts creates conditions for recontextualising auto-operative procedures. This brings the software designer on stage. Whereas software development is very often seen as related to software embedded in devices, software design puts emphasis on software embedded in social contexts such as human work and communication lending meaning to both tangible and intangible technological artefacts in a quite different sense. The plurality of contexts and the tensions stemming from the differentiation between internal and external fields of action becomes then crucial. But this is the central point of the next section. What we want to focus on here is the fact that social contexts make technologists less of a scientific problem-solver and more a sort of designer as we encounter them in other fields.

The critique of design fallacy combined with the shift from auto-operative to enacted uses of technology draws our attention to the need of understanding the designer communities as a priority compared to the understanding of user communities. What we would need is to go back to the internal world of software designers and see how they create their internal dynamics without tracing back every possible aspect to the complex elaboration of the external world or to the external influences inflicted upon fields of design practices. In other terms, it is because of the boundary created around these designer communities that they can create a point of view which allows them to shape the messages of the world as a result of their intentionality. It is their internal constitution which makes the world of software development meaningful to them. In other words, for various reasons related to the social organisation of design practices, there is no meaningful external view of the world without in-group dynamics or both real and imagined strong network ties within designer and developer teams. If there is a real significance of user orientation this has to do with representations leading to the social constructions of the user in designer communities.

The social organisation of software design

Software design is the progressive metamorphosis of knowledge concerning fields of action, and thus also concerning fields of social interaction, into a multilayered language that can be read and executed by a computer and in parallel understood by operators and users. Contrary to programming, software design is an exercise in manipulating resources in order to create not only executable instructions for a machine, but also to describe intellectual property, i.e. forms of knowledge acquisition, rather than mechanical properties (King 2005). The processes of knowledge acquisition inherent in software design processes relate internal design environments with environments of use external to these in a dynamic manner. Software development, and especially its design part, is not an introvert activity taking place in inward looking closed communities of practice or by remote and solitary actants. It relies on the interplay between internal and external social and organisational environments. This means that as a practice it cannot rely on social and symbolic resources of a totally marginalised community as is often the case with many research communities structured around stable central problems and paradigms. Software production is always the result of a diverse set of both strong and weak ties among members of various communities. In Ronald Burt's terms, it always takes place in structural holes of social networks (Burt 2004). One of the implications of this is that solutions and problems never coincide with technology, which means that there are strong reasons to look beyond the technical features for the constitutive aspects of such processes (Nyce & Bader 2002, 29). This coincidence of solutions with technology can be encountered only in cases where we have marginalised closure, when an almost autopoietic reproduction of internal symbolic and cognitive resources of communities suffices to bring about the expected products of both manual and intellectual labour. This is the case whenever software engineers tend to view these processes as a direct outcome of hard and supposedly ultra-rational decisions leading to the crystallisation of operational knowledge and to its mapping onto a completely unambiguous language (King 2005). However, both the origin in social representations and the complexity of interactions among the various layers of the language

code (machine code – programs – end-user application – operator instructions – users' information etc.) make software a contingently structured product. These contingencies and the indeterminacies they may imply could never be the basis of delivering IT products and services without the preceding shaping action of the charismatic software designer. How much charisma is needed depends on various factors, among these being the innovativeness of ideas they seek to implant in milieus of resisting users, or the intensity of change required in designing and developing teams.

Since creating and manipulating the contingent and multilayered language of software implies manipulating sets of both artefacts and humans in user contexts, software designers must bring about the symbolic framework and the rhetoric needed to make marketers and implementers of technologies succeed. This implies that a tight link between meaning and operation appears as a prerequisite of the life of any system created by the designer's mind. Consequently, without the appropriate cultural production any symbolic manipulation as part of the technical artefact is rendered useless (Brown & Duguid 2000). Software design, in trying to face this, becomes an act of putting objective realities, i.e. brute facts, in perspective. It is an act of transforming brute facts into institutional facts, and through this, sets of institutional facts into other sets of institutional facts. The intention of producing a perspective in networks consisting of both artefacts and humans is what counts. Software design is, in reality, more requirements determination than requirements analysis (Hohmann 1997, 12–13; 2003, 1).

The word perspective, although it derives from the usual metaphor, is meant here in a more literary sense. Erwin Panofsky wrote in his celebrated treatise published in 1927 that the 'Perspective in transforming the ousia (reality) into the phainomenon (appearance), seems to reduce the divine into subject matter for human consciousness; but for that very reason, conversely, it expands human consciousness into a vessel for the divine'. In more profane words, socially constructed perspectives transcend mundane things into meaning by putting them in the context of micro-fields. What makes software design so special is that, contrary to programming and software engineering, it brings about new instances in processes of the social organisation of technological communities of practice. Designers,

contrary to programmers and software engineers, transform arenas of development into arenas of design. What is it that makes the difference? The main thing is that designers, and for that matter software designers, seek to create and impose upon others new perspectives. Design (to paraphrase Adrian Forty, 1986, 11) and technological (software) design alter the way people see commodities, devices and systems. Design is critique triggering social change (Berg 1998). Further design is a form of recursive but directed conversation with objects and significant others (Fry 1999, 289–290). The directedness is driven by the imagination of designers facing prospective operational situations or situations of use. This means that inevitably the imagination of the designer is directed towards the creation of a community of practice constituted around the artefact emerging from the process of design. On the other hand, despite the momentum of intentionality and the orientation towards novelty, design action has its limits set by the world and the pre-shaped minds of the people and the features of the artefacts and the materials involved. This means that the designer's mind as well as the minds of users are always maladapted to the part of the world consisting of brute facts, or of institutional facts external to their own community, i.e. of facts not shaped by the design act. The designer overcomes this obstacle through mythologies, which enhance both his / her rhetoric and design methodologies. This is of decisive importance for the way practices of software design are embedded in forms of social organisation with specific characteristics.

Software designers never work alone. Software development processes, in which software design activities take place, are complexly structured multifaceted and multifunctional group processes comprising outcome reviews, project meetings, informal exchange, both vertical and horizontal, of information on tools, critique of products and services etc. (Hohmann 1997, 149–150). Groups of software developers and designers must often accommodate many different specialists and stakeholders and deal with the uncertainties which the interactions among them may cause. Solutions in such environments emerge from a perpetual state of discovery, not only material, but social discovery as well. The fields of action involved create many different outcomes which try to reduce uncertainty and increase understanding. Within these processes software designers create blueprints, scale

models, describe computer-based solutions, and produce requirements documents, data models, prototypes and charts originating from a plethora of decisions (Hohmann 1997, 24, 34–35). However, the complexity of these processes is due rather to social than to artefactual factors. A high percentage of the members of the social groups undertaking such activities are really charismatic personalities; many others believe that they are or imagine that they could be. This does not apply only to the legitimate leaders in such social groups, but also to many other specialists who have to accomplish even minor creative tasks and thus must maintain a pride in their work if they want to keep being productive.[8] Within the software developing teams designers are not the only ones who feel like prima donnas. This may apply to a lesser extent to systems or business analysts or requirements engineers. But programmers are incurable individualists.[9] This relatively strong ego, which often emerges from the necessity of distributed knowledge and creativity, is not always helpful. It can happen that in the end individual group members have to solve problems on their own, but there is a constant shuttle between working alone and working with others, as there is a fluid alternation between relative hierarchical positions depending on the switch between various subtasks. Code writing and programming can be crucial here and cause forms of both cooperative and non-cooperative games of social interaction. Further, tensions amongst various specialities, with the most characteristic ones being between analysts and programmers, but also amongst other stakeholders, must also be tackled (Bødker et al. 2002, 280; Nørbjerg & Kraft 2002, 211–213). One of the problems chief software designers have then to face is that establishing hierarchies in teams with a complex system of division of labour and dependence on horizontal communication amongst knowledge workers is much more difficult than in other kinds of designers' workshops. Software designers, if they want to see their ideas transformed into deliverables, must shape the teams of co-designers and developers. This is something that all designers depending on larger groups know, namely that there is no design if you cannot appropriately design and cultivate forms of collaboration in your own team. Orientation towards qualitative standards, as well as styles of design and consequent features of products depend very much on styles of the internal organisation of designer teams.[10]

Designing and setting up a design and development team means, among others, establishing commonly accepted forms of interaction and relevant rules, sometimes on the basis of established or in-house methodologies. Established generic methods often related to IT products and services with a strong brand, as well as local proprietary methods and requirements documents, undoubtedly play a strongly influential role in these decisions. Nevertheless, subjective and inter-subjective criteria become more important than the supposedly objective technological ones.[11] Value orientations and emotions become much more important and are also much more easily articulated whenever explicit and pressing externally imposed requirements do not exist or are loosely defined. Cognitive styles related to social and cultural codes, which describe how much externally imposed structure we want to afford during problem-solving, can also be crucial. Social goals capture the imagination of technologists and provide them with preferences for tools and both material and symbolic aspects of practices. For example, the goal of certain group members might be to become proficient in Java™ or to be rather a fashionable C++ type of programmer than an old fashioned but indispensable COBOL specialist, or to acquire prestige and status in a group through innovative knowledge and unprecedented skill. However, time criticalities and the perception of operational risks to which projects are exposed can significantly undermine these aspects or the tendency towards relatively autonomous subjectivity. Top-down approaches can then appear justified. Deadlines and risk perception can bring rigid method and command lines back in, or at least deliver a pretext for the instalment of managerial power. The more deadlines or standards-related pressures are encountered in projects, the more the rationale for the importance of method is related to the idea that its use facilitates coordination and thus legitimates the coordinator's roles and power.[12] A method such as those deriving from familiar system architectures, or from a topology of system objects, provides a starting point for high-performance teams (Hohmann 1997, 178). This can succeed because of the role distribution and the hierarchies deduced and justified on the basis of a reference to the strength of a method—subsequent technologies of the self do the rest and enhance compliance of the ones who are placed at lower echelons within the group. Software designers and software

developers may try to transform a situation not very convenient for the exploitation of creative intentions and productive skills into one with much higher degrees of freedom. However, opportunities and contingencies vary significantly and under certain conditions they tend to shrink to zero. This is, for instance, the case when hierarchical pressures combined with division of labour or segmentation of project components lead to sets of prescribed tasks. The types of project can also play a role. There are things like company bids, public tenders, subcontracting, outsourcing, insourcing, parameterisation, customisation, etc. which can restrict the creative element down to subsystem or component level, or even worse, to the level of building components of device-embedded auto-operative software or of collecting data for presales reports. At this level and in these practical domains, nothing more than the design of technical aspects is possible, a design which does not include design of social aspects of technology (relations of configurations of artefacts with configurations of users; communications streams, roles and emotional configurations, symbols and values, hierarchies etc.)—operational and instrumental features overwhelm all the others. This kind of technical design in its most sophisticated version can be compared with the application of existing knowledge to the solution of a practical engineering problem. It rests on disciplinary knowledge in the sense of normal science, and the social prestige of the bearers of such expert knowledge derives from the repetition of invariant information, as well as from the reduction of knowledge to constant principles and maxims. This pressure often influences designing communities because of the centrality of the marketing function. Not only technology per se and its scientific rationalisation, but also the more banal impact of the marketing function upon software design can have significant consequences in the direction of standardisation. Ritual and hypocrisy in language used with the purpose of appearing conventional to managers and clients can be one of the possible results. Speaking normatively but acting informally and perhaps also non-institutionally is one of the tensions which characterise IT communities—and this occurs the smaller the social distance is to user organisations.[13] Social proximity to programming and hard-core software engineering communities with high prestige and an intimidating scientistic posture can also play a role. Hard-core engineering

communities can be very successful in imposing tools, linguistic standards and rituals. Disputes on the modalities of integration in systems of interacting humans and artefacts can be another issue related to power struggles in or among teams. Pressure for integration also means pressure for better communication, knowledge and information management and common understanding of problems and problem-solving strategies on the ground of the acceptance of perceived technological parameters (Hohmann 1997, 235). But although structure, formalisation and institutionalisation are needed, they cannot be achieved on the basis of ritualism. As we have already stressed, these rules and methodologies can never be as stable as many would wish. Improvisation and improvised coordination can here be of decisive importance as a way to realise or to complement rules. Software designers as managers in these contexts have to encourage innovation and thus create spaces of freedom and 'creative flow', but at the same time create spaces of containing and disciplining others, otherwise projects and plans 'can spin out of control' (Cusumano 2004, 3). This does not work with factory mentality.[14] As Cusumano (2004, 130) observes in his study on Microsoft: 'It is unreasonable to expect most software projects to have both successful outcomes and be easy to manage'.

Software designers always belong to conflict laden arenas of design. In other terms, the social organisation of arenas of design is what makes software design possible, whereby social organisation is not understood as something stable and harmonious.[15] We understand social organisation here as a mapping of social orders (induced or spontaneous) upon forms of action. Although it is a truism to say that social action is possible and thinkable only in the framework of social organisation, it is not equally obvious that this implies contingent non-arbitrariness. Social action takes place in social orders which regulate in a contingent, not in a deterministic manner, both content and form of purposive action and the direction of intentionality. Social action is then a constant reordering both in the sense of network structures and mental maps emerging from the intentionality of actants. Coexistence, coordination and collaboration is then the outcome of fluid and selective ties on the basis of changing configurations of value orientations, as well as on the basis of changing configurations of forms of exploitation of material, social and cultural

resources. These changing configurations are the outcome of changing configurations of intuitions and interpretive discursive utterances. This relatively fluid and although not always conscious, but definitely active elaboration of social orders makes social organisation also something different from social structure. Firstly, actors are not compliant conformists and this structure does not fully determine action. Secondly, structural constraints have an indirect impact by creating the contingencies in spaces of action. These contingencies imply forms of resistance, translations and inversions of symbolic orientations and meaning. In this sense structural frameworks can tacitly imply or even sometimes explicitly provoke, through uttered interpretations or 'interpellations' addressed to involved actants, processes of transposition of their own substance. In this sense structural moments can function as regulatory principles of contingently induced or directed changes in the form of social organisation. Decisions as radical acts of structural detachment always derive from the history of structures. Control by powerful actors also makes sense on the basis of structural consciousness. This also applies to escaping control through social differentiation, as well as attempts to stabilise forms of social interaction through institutionalisation or formalisation on the basis of language and rituals. A script describing roles can be one of the main aspects of such developments. Such role ascription or role achievement can be related to changes in both central and instrumental social values. The centrality of values and the degree of implication for encountering problems of structural anomy can be of decisive importance for the understanding of the degree of innovation and change. What is important for the discussion here, however, is that innovations, and embodied innovators, also change the forms and content of social organisation, i.e. innovators always try to bring about, in various degrees, social change. And this applies also to technological innovators to whom software designers belong.

Arenas of software design are not socially organised in the same way as arenas of mere software development, although there can be no arena of design without development functions.[16] As their social organisation emerges from intentions to adopt action enabling the accommodation of innovators, arenas of design are much more laden with emotions stemming from an active encountering of social, organisational, technical and material

constraints and opportunities than are conventional and reproductive arenas of development. Actors in arenas of software design are interested in cultivating their ego through novelty and thus tend to be characterised less by reproductive and more by productive skills. Furthermore, software designers are innovators who provoke change of the social organisation not only in contexts of use accepting the products of the design, but also in the arena of design to which they belong. This means that arenas of design are in constant flux, contrary to arenas of more conventional development, which can rely on the reproduction of results of previous discoveries, inventions and successful promotion of past innovations. The social and psychological profiles of software designers correspondingly diverge from those of more conventional software developers. The dual role of software designers, who must be at the same time innovators of contexts of use as well as of contexts of design, can be taken on only by those members of modern societies whom Margaret Archer characterises as 'autonomous reflexives'.

Autonomous reflexives are subjects who possess the ability to describe and reshape their own identity as a consequence of the deliberate intention to reposition themselves in processes of social organisation. In addition, they can distance themselves from structural and symbolical constraints by inventively using and reordering inter-personal relations (Archer 2003, 133, 188, 227–228). Software designers, as autonomous reflexives, cultivate their roles through the appropriate narrative, i.e. they possess the ability to create their own myths endorsing the invention and perhaps also the consecutive re-inventions of their identities, as well as of the identities of the ones they seek to organise according to their plans.[17] Positioning software designers among autonomous reflexives makes it easier to explain how we propose to understand their arenas of activity. As arenas of design are considered here as social fields in which we can find forms of social organisation, or are part of forms of social organisation, depending on the existence in them of autonomous reflexives, collaborating with other less reflexive individuals, with operational intentions to produce (tangible or intangible) artefacts.[18]

Micro-sociological perspectives

The adoption of micro-sociological perspectives of software design processes does not necessarily imply ignoring the macro-processes. Socio-technical regimes, the formation and evolution of transnational scientific disciplinary networks, economic parameters, the political system and legal regulations, as well as public opinion and widespread cultural patterns can be of decisive importance for the macro-environment of social clusters in which software design and development take place. Macro-processes define the availability of resources and constraints, and this plays a role especially when micro-fields attract the attention of powerful actants who are interested either in controlling and exploiting activities or in preventing the emergence of antagonistic groups. But all these aspects are rendered meaningful through their translation into micro-processes where empowered subjects have more opportunities to resist the mystification of social relations (Scheff 1990, 188). Action can be understood as discursively embedded only if localized in small groups and demarcated fields of social organisation. Macro-structures, without being elaborated by human consciousness in micro-fields, do not acquire any meaning, and thus work just like dead architectures. Yet studying forms of the organisation of micro-fields through social interaction presupposes a linkage between interaction and memory (Harrington & Fine 2006). Forms of social interaction have their own temporality which makes them meaningful through reference to the location of events in the past, the present, and the future. In the case of software design teams, the future is obviously related to the designers' plans and the present is perceived in relation to technological objects, like electronic devices, manuals, tools, inscription devices, recorded methodologies etc. The past is related to memory and thus to the deeper layers of emotional states originating in the tensions between individual desires and the necessities of collective action. The design of futures, and of course in this case of future technological artefacts, mobilises both individual and shared pasts of group members. This is not always a harmonious process. Relations to others, as well as relations to objects of use and objects of design, result in sequences of positive and negative emotions with often diverse consequences among

members of a group. This provokes an undulation between remembering and forgetting as part of social interaction. Software designers as autonomous reflexives are especially active in managing this selective remembering or forgetting. Their collaborators with lower social prestige tend rather to forget than to remember, depending on the frequency of traumatic situations due to consecutive pressures for suppressing parts of their identities and desires. Forgetting can then be the constitutive element of technologies of the self.

Conditions of social organisation and situations which autonomous reflexives desire or try to avoid, create orientations and emotions which have the potential to change themselves as well as the ones who see them as significant others either by compulsion or by identification. The direction of deliberate change or the mechanisms of resistance to change which cause unintended consequences of action rely on a consciousness which is always intentional. This is a consciousness about objects constructed through abstractions emerging from the dominant discourses in socially organised contexts of interaction. These dominant discourses can stem partly from the surrounding macro-processes, but mainly from the translations of methodologies and guidelines appearing in the leading designers' plots. Such socially generated abstractions require substantial omissions and forgetfulness. Omitting and forgetting is what creates the modalities of interaction which allow arenas of software development to become organised. Representations, artefacts, tools, inscriptions and all the interacting elements involved in the design and development of products and services change because of the emergence and subsequent transformation of emotions related to omitting and forgetting. Only when technological objects are felt as emotionally relevant can they contribute to the emergence of movers of action; and this is also the case when they are viewed as mere constraints and not as resources enabling the spotting of opportunities.

The idea of forgetfulness as part of processes in groups oriented towards exploiting knowledge for the shaping of action, as is the case with designing teams, also draws on the late Husserl, especially on his book on the crisis of the European sciences. He sees the problem of crisis there as a pulling apart, as the articulation of a process of separation. The separation he is mainly thinking of is that caused by the distance we take

from the way we have separated ontological entities which concern us at an earlier stage. This is forgetfulness as suppression of a past act of tearing entities apart. This is what we do when we adopt and internalise guiding methodologies. Their effective application presupposes forgetting our original feelings in order to avoid reflexive disturbance of actions and discourses considered as obvious and necessary (Ciborra 2002, 15). This parting of the structural imagination, stemming from the externalisation of the angle of observation, from the consciousness and the feelings of the ones involved in demarcated fields of action is the main problem we are facing whenever we try to understand the evolving social organisation and the intricacies of meaningful purposive action in fields of software design. By adopting a supposedly scientific method we tend to disregard the fundamental importance of the everyday world of the technical staff, the designers, and the managers. We tend to look away from the messiness and situatedness of the enactment of practices, while privileging descriptions on the basis of geometric worlds created by system methodologies. The idealities of methods incline us against empirical reminders which can trace the process of abstraction back to their emotional origins.

We usually think that whenever software designers, software developers, software engineers and programmers use technical and ritualistic language, they are sterile technocrats without feelings and emotions. But this must be regarded as a signal for exactly the opposite. It means that they are construing their world as technocratic and find this a convenient solution in order to manage their emotions. In arenas of design there is a moral content of artefacts and procedures: they produce the justification of practices, as well as the emotional framing of cognitive orientations. However, if we want to talk in Heideggerian terms, this is hidden behind forms of objectification stemming from the 'technological mood' of software designers as a way of being oriented towards technological solutions, i.e. towards ordering the world around us as the only way to create meaning in arenas of development. How can we discover the substance of this technological mood by going behind the contradiction between the technological and the pre-technological self? According to Andrew Feenberg (1999, 183–199), situations of technological mood also contain non-technological moods, or in other words, alternative expressions of the technological self.

Designers who are inclined to put artefacts and methodologies in perspective adopt this attitude in particular. This means that although modern technology frames the world for us as 'devices' and hides the full referentiality (or contextuality) of the world upon which they depend for their ongoing functioning, we can very often escape from this even by using the very language of technology. Artefacts cause emotions as they are encountered by those who take part in interaction ritual chains in contexts of design.[19] They are part of their social world and the way people speak about them creates emotions and influences action.[20] As Ernst Cassirer has stressed in his philosophy of symbolic forms, objects exist for the self only through the affective spark they may provoke. Otherwise they are neither perceived nor acquire meaning. This implies that the signal for triggering any form of deliberating planned behaviour, as is the case in contexts of design, stems from the creation of such emotional settings through interaction ritual chains. Designers, as many other creators too, are obliged to understand objects and situations as a function of configurations of egos and try to influence them on the basis of a hierarchy of relevant themes. As Schütz points out, there are various counterpoised themes and horizons out of the configuration of which intentionality selects hierarchies of relevancies (Schütz 1982). When themes and horizons become unconscious or irrelevant, the problem is, following Bergson, not what we are remembering, but what and why we are forgetting. Forgetting in this case, however, should not mean only having difficulties in recalling something, but also spontaneously not paying attention, ignoring or not focusing on something. Sometimes we do not really forget but we make up our mind in favour of a hierarchy of utterances and concealments. Things we forget, we want to forget or we do not want to mention in spite of many indications appearing in a field which show that they are somehow there, are the most important ones for the constitution of dominant discourses and practices in a demarcated field of action. The micro-sociological perspective of software design process should drive the researcher not to pay attention to what is being said and done, but to the systematic omissions. The rituals of conscious concealment or the ones provoked by forgetting are perhaps the most decisive for understanding the high points of collective experience which, according to Collins

(2004, 42), render interaction ritual chains central instances of social organisation. In technological design contexts the presence of technological objects is crucial. Objects in the form of artefacts or methodological prescriptions can be devices of draining and relocating communicative energy. Any such direction of energy creates a rest, a complementary space, where unarticulated emotions are located. If they can be made manifest, then we can obtain the valuable concealed information.

Notes

1 The epistemological line along which such studies are made can be found in a chapter by Madeleine Akrich (1992) in a widely read book on the social shaping of technology edited by Wiebe Bijker and John Law.

2 Software design methods are based on ideas previously developed for program design methods. The hardest of these are based either on mathematical methods or on computer assisted software engineering (CASE) (Iivari 1996). Others draw on attempts to formalise architectural design as is the case with Christopher Alexander's pattern language. Modelling methods and techniques, especially those based on inscription devices (Latour & Woolgar 1986) or on formal, semiformal, or diagrammatic notation, as part of a systematic design phase during which stakeholders seek stabilisation of semantics (Jirotka & Luff 2002, 131) can be crucial and take up much of the intellectual energy of software designers and developers.

3 As Bertelsen and Bødker (2002, 412) have stressed, design is mediated by design artefacts. Thus 'Software designers and developers define design and development as a scientific, or as an engineering problem, and this is taken to be the natural order of things' (Nyce & Bader 2002, 31). On the origins of this in mathematical cultures: see Heintz (1993); MacKenzie (2001).

4 Customisation, parameterisation and versioning (the latter meaning putting unfinished IT products on the market provided that they will be upgraded by the vendor at a later stage) are characteristics of informational goods and IT services. For a discussion on this see Shapiro and Varian (1998).

5 Experts are supposed to immediately recognise what the hard part of the problem is and go directly into producing the corresponding solution (Hohmann 1997, 21). But as Scheff (1990, 19) points out, this makes them deny the social bonds and the emotional origin, or any other soft aspect, of choices and decisions. This hinders them from understanding and managing messes and orientates them towards rigid definitions of problems, which implies in most cases deductive

approaches to solution finding and solution design. In other words experts very often lack the ability to think and act in dialogue with complex situations, unstable information and fluid configurations of resources, i.e. as 'reflective practitioners' (Schön 1983).

6 Software is a cultural object, the emergence of which cannot be traced back to any formal aspects of mathematical calculus. Even computer code becomes cultural in the process of production, circulation and consumption of information which imply the mediation of informatic sociality resulting from the intersections between the different sets of practices and conventions. As Adrian MacKenzie (2005) has shown in the case of Linux, code objects emerge from collective agency especially in open source software development. This is in fact not something new, as we have always encountered such processes either in intra- or in inter-organisational behaviour. The difference in this case is more related to issues of property rights.

7 Formalisms are supposed to help to remove the ambiguity of incomplete descriptions and help designers to achieve common knowledge. However, we have more common talk than common knowledge. As Luciana D' Adderio (2004, 45) points out, software design deals with open or incomplete descriptions of the world. The key problem for all software designers is balancing the informal and formal assumptions. This balance is by no means easy, as designers are often socialised on the basis of literatures presenting a normative framework of design as a highly abstract activity following rules and patterns of well defined procedural steps (D' Adderio 2004, 45). This can reify the ambiguity and uncertainty of real processes emerging form fluid configurations of acts situated in art, science, engineering and tinkering which are difficult to control following rituals and methods. Squeezing everything into auto-operative procedures is how computer people prefer to handle real life, but in most cases it does not work like that: this internalisation, this enclosure in demarcated and self-reproductive codes without external interference is not what happens in real life, where we have to cope with various actors and stakeholders acting in complex social environments, as well as with various forms of communication, negotiation, institutional compromises and coordination among these (Andelfinger 2002, 190, 199, 201; Cusumano 2004, 149).

8 Software designers, famous stars of the guild excluded, do not enjoy wider publicity. They are known and recognized as important only in relatively small circles. Only a word of mouth reputation enables these people to survive psychologically and perhaps have interesting careers in the intra- or inter-organisational IT circles (Cusumano & Selby 1998, 63).

9 For those individualists, and in these kinds of organisational environments, knowledge sharing also gets difficult (Cusumano & Selby 1998, 71). The differentiation and segmentation of skills, the diverging non-cooperative prestige acquisition

strategies and the subsequent emphasis on discourses focusing on soft skills make the management of such teams much more difficult (Cusumano 2004, 171; Hohmann 1997, 155).

10 For instance, as Cusumano (2004, 11) points out, European software producers place more emphasis on elegant solutions than on marketing and sales, which can be interpreted as a rather introvert organisation of design spaces. Richard Coyne draws attention to a quite different dimension, namely to the role of 'techno-romanticism' as the constitutive element of organisational and network cultures. Neo-romantic cultures and 'digital narratives' emerging in communities of technological practices, like hackers and computer freaks, make software design and development practices more an artisanal than a factory like exercise (Coyne (1999, 12, 26–28; also Castells 2001, 36–63). Donald MacKenzie (2001, 333–334) stresses the differences between managerial and technological styles of approach which can have an impact on software design cultures. The difference can be traced back to whether personnel involved in software and information system development are working 'hands-on', and thus know better the risks as implied by the artefacts they produce, or to a more distant 'black-box' approach which evinces greater certainty but does not have insight into the limitations. These are parameters on which we can draw in order to better understand the impact of the orientations and composition of designing teams.

11 Model building, language creation and writing messages are acts of codification and objectification pursued in the knowledge environment where the interplay between tacit and explicit elements can destabilise environments of design (D'Adderio 2004, 16–17; Wong & Radcliffe 2000). However, objectification in the form of standardisation can have opposite effects because it creates problems of 'noise' across organisational boundaries due to organizational heterogeneity which hinders co-ordination across different communities on the basis of common knowledge.

12 Deadlines cause segmentation of social relations, because those who are spatially segregated in distant communities of practice and on which pressure is exerted through tight deadlines, tend to develop forms of in-group solidarity with aggression directed against the distant deadline setter (e.g. headquarters). This effect can reduce the likelihood of effective communication and genuine cooperation between sites (Nørbjerg & Kraft 2002, 215–216). Deadlines thus also contribute to the creating of conditions of asymmetric information and segmentation of practices—if processes are spatially differentiated.

13 Entrepreneurial practices can also be crucial in this respect. Contrary to designers who are interested in novelty and heroism, vendors are interested in risk management and market orientation. However, in the case of in-house developers, compliance is required, and at the end of the day what counts is that 'nobody

has been fired because of IBM'. But behind the adoption of branded products and services, tinkering is done by the unknown IT heroes, who make themselves important in small communities and in fragmented publicities, which the top management and the inter-organisational publicity do not perceive.

14 Structured programming was proposed as a way to cope with this contradiction. Structured programming is interesting because, in a way we have similarly observed in scientific management, the scientific background of engineering the artefact is projected upon the human aspects of the process of development and production. The reference to scientific principles offers the rationale for organising and managing people in well specified systems of division of labour under the guidance of 'chief programmer teams' (MacKenzie 2001, 39).

15 Raymond Firth's social anthropology was founded on a theoretical ground of elaboration of this idea of social organisation. Although his stance was, as in most structural-functionalist approaches, harmonistic and did not include the idea of contingency, or any closer analysis of social interaction, an inspiration from his central ideas can be extremely useful in this context (Firth 1969). What is also extremely useful from Firth's ideas, despite his functionalist obsessions, is that these either visible or invisible mappings of order on action draw on social targets and values. Furthermore, we can find in Firth the idea of co-ordination as part of the idea of social organisation. Although this is in my view the weak point of his theoretical construction, it provokes a counterpoised idea of conflict and negotiation.

16 The concept of the arena of development has been discussed by Jørgensen and Sørensen (1999). Although these authors take into consideration the possibility that arenas of development can incorporate actors that are excluded by the dominant translations (stemming from methodologies, techniques and standardizations), they do not try to tell us much about how this can emerge in socially organized fields of inter-action. This is due to a rather uncritical adoption of Actor Network Theory on the basis of which they discuss how representations, inscriptions and interactions transform objects into IT products and services. The links between 'distinct strategies and processes that link together both competitive and co-operative types of relations between locations inside and between actor-worlds' are taken for granted without questioning their social emergence in fields of social organization. This halves the value of their truly interesting approach, as it remains static and does not differentiate between the impact of conventional and innovative tools, inscriptions and representations as parts of actor networks.

17 Autonomous reflexives in the software industry can gain the prestige of heroes, the characters of which can be nourished from 'technoromantic' digital narratives on inventors, innovative software writers, techno-frontier artists and techno-creatives, computer visionaries, elegant hackers, multimedia wizards etc. (Coyne 1999, 31). The

constitution and change of communities of technoromantic software designers emerge to a great extent from such 'digital' narratives, sometimes lending a dramatic tenor to stories about information technologies (Coyne 1999, 7–9). However, in larger organisations such narratives tend to be bound to rather conventional symbolisms, which deprive them of their more expressive and emotional elements (Gabriel 2000).

18 Social aspects, of course, are not the whole story. Arenas of design are socio-technically structured. Network technologies and network economies have made this socio-technical structuration of arenas of development rely more and more on a combination of physical and virtual aspects, as is characteristically the case with open source software development. The virtual elements do not of course stand alone: the virtual and the physical presuppose each other and this implies a mediation of action and discourses through the electronic and virtual spaces (Sassen 2004, 78, 80–81). Further, in arenas of design there is a dynamic interconnection between actors, artefacts and standards which populate the arenas. Standards in particular create pressures to comply with constraints which originated from fields of action structured beyond the borders of the arenas of design (Kallinikos 2004, 141).

19 According to Randal Collins (2004, xii), 'an interaction ritual is an emotion transformer, taking some emotions as ingredients, and turning them into other emotions as outcomes'.

20 Objects in contexts of design emerge from non-objectified situations, i.e. from configurations of framings emerging from drifting deliberations of designers. This statement makes the view presented here different from similar observations by John Law (2000) who stresses that 'an object is an object so long as everything stays in place. So long as the relations between it and its neighbouring entities hold steady'. 'And it holds together, it is an object, while those relations hold together and don't change their shape' (Law 1999, 4).

References and further reading

Akrich, M. (1992), 'The de-scription of technical objects', in Bijker, W. E., and J. Law (Eds.), *Shaping Technology / Building Society. Studies in Sociotechnical Change*, Cambridge, MA: The MIT Press, 205–224.

Andelfinger, U. (2002), 'On the intertwining of social and technical factors in software development projects', in Dittrich, Y., C. Floyd, and R. Klischewski (Eds.), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press, 185–203.

Anderson, R. J. (1994), 'Representations and requirements: The value of ethnography in system design', *Human-Computer Interaction* 9 (2): 151–182.

Archer, M. (2003), *Structure, Agency and the Internal Conversation*, Cambridge: Cambridge University Press.

Berg, M. (1998), 'The politics of technology: On bringing social theory into technological design', *Science, Technology, & Human Values* 23 (4): 456–490.

Bertelsen, O. W. (2000), 'Design artifacts. Towards a design-oriented epistemology', *Scandinavian Journal of Information Systems* 12 (1): 15–27.

Bertelsen, O. W., and S. Bødker (2002), 'Discontinuities', in Dittrich, Y., C. Floyd, and R. Klischewski (Eds.), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press, 409–424.

Bødker, K., F. Kensing, and J. Simonsen (2002), 'Changing work practices in design', in Dittrich, Y., C. Floyd, and R. Klischewski (Eds.), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press, 267–285.

Boehm, B., H. D. Rombach, and M. V. Zelkowitz (Eds.) (2005), *Foundations of Empirical Software Engineering*, Berlin / Heidelberg / New York: Springer.

Brown, J. S., and P. Duguid (2000), *The Social Life of Information*, Boston: Harvard Business School Press.

Bucciarelli, L. L. (1988), 'Engineering design processes', in Dubinskas, F. A. (Ed.), *Making Time: Ethnographies of High-Technology Organizations*, Philadelphia: Temple University Press, 92–122.

Bucciarelli, L. L. (1996), *Designing Engineers*, Cambridge, MA: The MIT Press.

Bucciarelli, L. L. (2003), 'Design and learning: A disjunction in contexts', *Design Studies* 24 (3): 295–311.

Burt, R. S. (2004), 'Structural holes and good ideas', *American Journal of Sociology* 110 (2): 349–399.

Castells, M. (2001), *The Internet Galaxy. Reflections on the Internet, Business and Society*, Oxford: Oxford University Press.

Ciborra, C. (2002), *The Labyrinths of Information. Challenging the Wisdom of Systems*, Oxford: Oxford University Press.

Cockburn, A. (1996), 'The interaction of social issues and software architecture', *Communications of the ACM* 39 (10): 40–46.

Collins, R. (2004), *Interaction Ritual Chains*, Princeton / Oxford: Princeton University Press.

Coyne, R. (1999), *Technoromanticism. Digital Narrative, Holism, and the Romance of the Real*, Cambridge, MA: The MIT Press.

Crabtree, A. (2004), 'Taking technomethodology seriously: Hybrid change in the ethnomethodology-design relationship', *European Journal of Information Systems* 13: 195–209.

Crabtree, A., and T. Rodden (2002), 'Ethnography and design?', in *Proceedings of the International Workshop on 'Interpretive' Approaches to Information Systems and Computing Research*, London: Association of Information Systems, 70–74.

Cusumano, M. A. (2004), *The Business of Software*, New York: Free Press.

Cusumano, M. A., and R. W. Selby (1998), *Microsoft Secrets*, New York: Simon & Schuster.

D' Adderio, L. (2004), *Inside the Virtual Product. How Organizations Create Knowledge through Software*, Cheltenham: Edward Elgar.

Dasgupta, S. (1991), *Design Theory and Computer Science*, Cambridge: Cambridge University Press.

Dittrich, Y. (2002), 'Doing empirical research on software development: Finding a path between understanding, intervention, and method development', in Dittrich, Y., C. Floyd, and R. Klischewski (Eds.), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press, 243–262.

Dittrich, Y., C. Floyd, and R. Klischewski (Eds.) (2002), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press.

Feenberg, A. (1999), *Questioning Technology*, London / New York: Routledge.

Firth, R. (1969), *Essays on Social Organization and Values*, London: The Athlone Press.

Floyd, C. (2002), 'Developing and embedding autooperational form', in Dittrich, Y., C. Floyd, and R. Klischewski (Eds.), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press, 5–28.

Forty, A. (1986), *Objects of Desire. Design and Society since 1750*, London: Thames & Hudson.

Fry, T. (1999), *A New Design Philosophy. An Introduction to Defuturing*, Sydney: University of New South Wales Press.

Gabriel, Y. (2000), *Storytelling in Organizations. Facts, Fictions and Fantasies*, Oxford: Oxford University Press.

Gal, S. (1996), 'Footholds for design', in Winograd, T. (Ed.), *Bringing Design to Software*, Boston: Addison-Wesley, 215–227.

Harrington, B., and G. A. Fine (2006), 'Where the action is: Small groups and recent developments in social theory', *Small Group Research* 37: 4–19.

Heintz, B. (1993), *Die Herrschaft der Regel. Zur Grundlagengeschichte des Computers*, Frankfurt am Main: Campus.

Heintz, B. (2000), *Die Innenwelt der Mathematik. Zur Kultur und Praxis einer beweisenden Disziplin*, Berlin: Springer.

Hohmann, L. (1997), *Journey of the Software Professional. A Sociology of Software Development*, Upper Saddle River, NJ: Prentice Hall.

Hohmann, L. (2003), *Beyond Software Architecture*, Boston: Addison-Wesley.

Ihde, D. (1990), *Technology and the Lifeworld: From Garden to Earth*. Bloomington / Indianapolis: Indiana University Press.

Iivari, J. (1996), 'Why are CASE tools not used?', *Communications of the ACM* 39 (10): 94–103.

Jennings, N. R. (2001), 'An agent-based approach for building complex software systems', *Communications of the ACM* 44 (4): 35–41.

Jirotka, M., and P. Luff (2002), 'Representing and modeling collaborative practices for systems development', in Dittrich, Y., C. Floyd, and R. Klischewski (Eds.), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press, 111–139.

Jørgensen, U., and O. H. Sørensen (1999), 'Arenas of development—A space populated by actor-worlds, artefacts, and surprises', *Technology Analysis & Strategic Management* 11 (3): 409–429.

Kallinikos, J. (1995), 'The Archi-tecture of the invisible: Technology is representation', *Organisation* 2 (1): 117–140.

Kallinikos, J. (2004), 'Farewell to constructivism: Technology and context-embedded action', in Avgerou, C., C. Ciborra, and F. Land (Eds.), *The Social Study of Information and Communication Technology*, Oxford: Oxford University Press, 140–161.

Kaptelinin, V. (2002), 'Making use of social thinking: The challenge of bridging activity systems', in Dittrich, Y., C. Floyd, and R. Klischewski (Eds.), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press, 45–68.

Kemerer, C., and S. Slaughter (1997), 'Methodologies for performing empirical studies: Report from the international workshop on empirical studies of software maintenance', *Empirical Software Engineering* 2 (2): 109–118.

King, D. (2005), *Parting Software and Program Design*, PhD-Thesis, University of York.

Latour, B. (1992), 'Where are the missing masses? The sociology of a few mundane artifacts', in Bijker, W. E., and J. Law, (Eds.), *Shaping Technology / Building Society. Studies in Sociotechnical Change*, Cambridge, MA: The MIT Press, 225–258.

Latour, B., and S. Woolgar (1986), *Laboratory Life: The Construction of Scientific Facts*, Princeton: Princeton University Press.

Law, J. (1999), 'After ANT: Complexity, naming and topology', in Law, J., and J. Hassard (Eds.), *Actor Network Theory and After*, Oxford: Blackwell, 1–14.

Law, J. (2000), 'Objects, spaces and others', http://www.lancs.ac.uk/fss/sociology/papers/law-objects-spaces-others.pdf.

Lawson, B. (1997), *How Designers Think*, Oxford: Architectural Press.

Lethbridge, T. C., S. E. Sim, and J. Singer (2005), 'Studying software engineers: Data collection techniques for software field studies', *Empirical Software Engineering* 10 (3): 311–341.

Luff, P., J. Hindmarsh, and C. Heath (Eds.) (2000), *Workplace Studies. Recovering Work Practice and Informing System Design*, Cambridge: Cambridge University Press.

Luhmann, N. (1991), *Soziologie des Risikos*, Berlin / New York: Walter de Gruyter.

McCarthy, S. (1995), *Dynamics of Software Development*, Redmond, WA: Microsoft Press.

MacKenzie, A. (2005), 'The performativity of code. Software and cultures of circulation', *Theory, Culture & Society* 22 (1): 71–92.

MacKenzie, D. (2001), *Mechanizing Proof. Computing, Risk and Trust*, Cambridge, MA: The MIT Press.

Nørbjerg, J., and P. Kraft (2002), 'Software practice is social practice', in Dittrich, Y., C. Floyd, and R. Klischewski (Eds.), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press, 205–222.

Nyce, J. M., and G. Bader (2002), 'On foundational categories in software development', in Dittrich, Y., C. Floyd, and R. Klischewski (Eds.), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press, 29–44.

Oberquelle, H. (2002), 'Userware design and evolution: Bridging social thinking and software construction', in Dittrich, Y., C. Floyd, and R. Klischewski (Eds.), *Social Thinking—Software Practice*, Cambridge, MA: The MIT Press, 391–408.

Panofsky, E. (1997 [1927]), *Perspective as Symbolic Form*, New York: Zone Books.

Sassen, S. (2004), 'Towards a sociology of information technology', in Avgerou, C., C. Ciborra, and F. Land (Eds.), *The Social Study of Information and Communication Technology*, Oxford: Oxford University Press, 76–99.

Scheff, T. J. (1990), *Microsociology. Discourse, Emotion and Social Structure*, Chicago / London: The University of Chicago Press.

Schön, D. (1983), *The Reflective Practitioner*, New York: Basic Books.

Schütz, A. (1982), *Das Problem der Relevanz*, Frankfurt am Main: Suhrkamp.

Shapiro, C., and H. Varian (1998), *Information Rules. A Strategic Guide to the Network Economy*, Boston: Harvard Business School Press.

Shedroff, N. (1994), 'Information interaction design: A unified field theory of design', http://www.vivid.com/form/unified/unified.html.

Singer, J., T. Lethbridge, N. Vinson, and N. Anquetil (1997), 'An examination of software engineering work practices', CASCON '97, Toronto, October, 209–223.

Sommerville, I. (2001), *Software Engineering* (6th Edition), Harlow: Pearson Education Ltd.

Sommerville, I., T. Rodden, P. Sawyer, R. Bentley, and M. Twindale (1993), 'Integrating ethnography into the software engineering process', in *Proceedings of the International Symposium on Software Engineering*, Los Alamitos, CA: IEEE Press, 165–173.

Stewart, J., and R. Williams (2005), 'The wrong trousers? Beyond the design fallacy: Social learning and the user', in Rohracher, H. (Ed.), *User Involvement in Innovation Processes. Strategies and Limitations from a Socio-Technical Perspective*, München: Profil, 39–71.

Vincenti, W. G. (1990), *What Engineers Know and How they Know It*, Baltimore / London: The Johns Hopkins University Press.

Wildbur, P., and M. Burke (1998), *Information Graphics. Innovative Solutions in Contemporary Design*, London: Thames & Hudson.

Winograd, T. (Ed.) (1996), *Bringing Design to Software*, Boston: Addison-Wesley.

Wong, L. P., and D. F. Radcliffe (2000), 'The tacit nature of design knowledge', *Technology Analysis & Strategic Management* 12 (4): 493–512.