

On The Human, Organizational, and Technical Aspects of Software Development and Analysis

Robertas Damaševičius

Abstract

Information systems are designed, constructed, and used by people. Therefore, a software design process is not purely a technical task, but a complex psycho-socio-technical process embedded within organizational, cultural, and social structures. These structures influence the behavior and products of the programmer's work such as source code and documentation. This chapter (1) discusses the non-technical (organizational, social, cultural, and psychological) aspects of software development reflected in program source code; (2) presents a taxonomy of the social disciplines of computer science; and (3) discusses the socio-technical software analysis methods for discovering the human, organizational, and technical aspects embedded within software development artifacts.

Keywords Socio-technical software analysis · Program comprehension · Information system development

1. Introduction

Software engineering (SE) is primarily concerned with developing software that satisfies functional and non-functional requirements, internal and external constraints, and other requirements for usability, compatibility, portability, reusability, documentation, etc. Such requirements reflect social and organizational expectations of how, where, when, and why a software system may be used. However, software developers are not influenced just by given requirements and constraints. The quality, structure, and other characteristics of the developed software systems also depend upon education of software designers and programmers, their work experience, problem-solving strategies [56], organizational structure and social relations, shared mental models [22], cultural traditions and nationality [6], worldview, religion (Eleutheros), and even such minor aspects whether a coffee machine is installed in their workplace [27]. The importance of non-technical factors in information systems development (ISD) is underscored by a survey [17], which claims that 90% of ISD project failures can be attributed to non-technical (social, organizational, etc.) factors. Therefore, a software design process is not purely a technical task, but also a social process embedded within organizational and cultural structures. These structures influence and govern the work behavior of programmers and their final products, such as source code and documentation.

The socio-technical relationships between programmers and their developed software are very complex to register and study and they cannot be replicated experimentally or described using formal models. The actions and environment of software designers are rarely directly available for study. In many cases the only available material for analysis is program source code. The knowledge gained from source code analysis despite its likely partiality and ambiguity can tell us about software design processes, its

Robertas Damaševičius • Software Engineering Department, Kaunas University of Technology, Kaunas, Lithuania

development history, and provides us with some information about its author. Comprehension of source code may allow us to understand what the original programmer had comprehended [18].

From the socio-technical perspective, the structure of software systems can be described in terms of technical relationships between software units (components, classes, units, etc.) and social relationships between software developers and their environment. By analyzing such relationships and dependencies, we can uncover and comprehend not just the links between programmers and their code, but also the relations between programmers through their code. Socio-technical software analysis [62, 63, 16, 28] tries to uncover these socio-technical dependencies by analyzing artifacts of software design processes.

The aims of this chapter are (1) to overview the psycho-socio-technical aspects of software design reflected in program source code; (2) to discuss the social disciplines of computer sciences; and (3) to focus on socio-technical software analysis methods for discovering psycho-socio-technical aspects embedded within it.

2. Non-Technical Aspects of Software Development

2.1. Social Aspects

ISD is a socio-technical process [45, 57], which is affected by personal [13] and group [68] factors. Sawyer and Guinan [58] even claim that social processes had more influence on software quality than design methodologies or automation. There is considerable evidence that software design processes are influenced by social and psychological factors [30, 4, 27, 47, 7, 19, 10, 57]. The social nature of software development and use suggests the applicability of social psychology to understanding aspects of SE.

Programmers do not exist in isolation. They usually communicate about technical aspects of their work. Several studies [25, 62] suggest that technical dependencies among software components create “social dependencies” or “networks” among software developers implementing these components. For example, when developers are working to implement a software system within the same team, the developers responsible for developing each part of the system need to interact and coordinate to guarantee a smooth flow of work [65]. Inevitably, during such coordination and communication, the designers are influenced by each others’ domain knowledge, programming techniques, and styles. Such influence can be uncovered in software repositories and found in the structure of the software artifact itself [63]. Therefore, software development (certainly at a large-scale) can be considered as a fundamental social process embedded within organizational and cultural structures. These social structures enable, constrain, and shape the behavior, knowledge, programming techniques, and styles of software developers [27].

2.2. Organizational Aspects

Other socio-technical aspects that influence the work of software designer are organizational aspects (e.g., structure of organization, management strategy, business model). Such dependence is often formulated as Conway’s Law: “organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations” [11]. There are numerous interpretations of Conway’s Law [29, 2]. In general, Conway’s Law states that any piece of software reflects the organizational structure that produced it. For example, two software components *A* and *B* cannot interface correctly with each other unless the designer of component *A* communicates with the designer of component *B*. Thus the interface structure of a software system will match the structure of the organization that has developed it.

Parnas further clarified how the relationship between organization and its product occurs during software development process. He defined a software module as “a responsibility assignment” [53], which means that the divisions of a software system correspond to a division of labor. This division of labor among different software developers creates the need to discuss and coordinate their design efforts [29]. Therefore, the analysis of software architectures can allow us to make conclusions about the organizational structure and social climate of the software designer’s team.

2.3. Psychological Aspects

Software design decisions are often based on psychological rationale, rather than purely computational or physical factors [69]. Software developers frequently think about the behavior of a program in mental or anthropomorphic terms, e.g., what a component “knows” or is “trying to do,” rather than formal, logical, mathematical, or physical ones [68]. About 70% of software representations are metaphorical [19], representing system behavior as physical movement of objects, as perceptual processes, or anthropomorphically by ascribing beliefs and desires to the system [30].

Software architecture is commonly considered as a structure of a software system. However, software architecture can also be analyzed as a mental model shared among software developers [51, 8, 33]. Mental models are high-level knowledge concepts of a designer that reflect both domain system structure and functions, software goals, design tasks, implementation strategies together with social, organizational, and psychological aspects that influenced the development of this system [31]. Uncovering and analyzing a mental model of a program is as important as analyzing formal or abstract models, and contributes toward a more comprehensive understanding of software development processes [44].

2.4. Cultural Aspects

To fully understand the relationship between software and programmers’ thoughts and actions, we need to understand the relationships between software and programmer culture [52]. The improvement of software design processes requires understanding of cultural context, practices, and sensitivities that these processes may relate to [60]. For example, although the need of transition to a new technology to improve the process of developing quality software products is well understood, new technologies are often adopted as a “silver bullet” without any convincing evidence that they will be effective, yet other technologies are ignored despite the published data that they will be useful [70]. Cultural concepts such as *postmodernism* [50] migrate to computer science and affect software development methodologies (e.g., claiming that there are no longer correct algorithms, but only contextual decisions), emphasizing code reuse, glue programming, and scripting languages.

For a long time the adaptation of software to a different culture has been only focused on the user interface (software internationalization). However, there is a growing trend that not only software, but also its development processes and methodologies need to be adapted [40]. Methods that were thought to be “best practices” for some cultural groups turn out to be ineffective or very difficult to implement for software developers from other cultural groups [6]. Nationality also has the influence. For example, one study [6] shows that American teams are culturally well suited for iterative development and prototyping. The same study asserts that American teams seem to enjoy the chaos, often to the detriment of project progress, whereas Japanese software project teams are more suited toward waterfall development styles.

3. Social Disciplines of Computer Science

The interdisciplinary nature of aspects related to software design has led to the arrival of social branches of computer science. The taxonomy of such branches is proposed in Table 2.1. The main object of research in these disciplines is the virtual world of software and its relationship with the real world of designers and users. These disciplines reflect a multitude of diverse views toward software as an object of creativity, medium of communication and idea sharing, and an evolving entity with its own life cycle and habitat.

When people program they express a philosophy about what operations are important in the world [21]. Common propositions such as “The best way to develop software is to use object-oriented design methodology” and “GOTO statements are harmful” relate as much to SE methodologies and programming language semantics as to philosophy and epistemology. *Software epistemology* [32, 38], which analyzes forms and manifestations of knowledge in software, examines the truthfulness of these and other propositions based on authority (expert’s opinion), reason (using rules of deductive logic), and

Table 2.1. Taxonomy of social disciplines of computer science.

| Discipline | Object of research | Aims |
|---|--|--|
| Social informatics (software sociology) | Uses and consequences of IT that takes into account their interaction with institutional and cultural contexts | To understand the relationship of technological systems to social systems |
| Software archeology | Legacy software or software versions | To recover, analyze, and interpret legacy software artifacts including source code, documentation, and specifications |
| Software axiology | Value and quality of software | To analyze formal methods for evaluating software value (formal software axiology), to analyze principles of visual beauty and appeal of software (software aesthetics), and to propose elegant programming methods and techniques |
| Software ecology | Software environment | To analyze methods for prevention of software pollution |
| Software epistemology | Forms and manifestations of knowledge in software | To analyze interaction between belief, intention, justification, and action that occurs in complex software systems |
| Software anthropology | Practices of software engineering | To analyze the day-to-day work of software engineers in the field |
| Software gerontology | Software aging processes | To analyze software aging, its causes, and prevention |
| Software morphology | Forms and shapes of software representations | To analyze the relationships between various parts of software |
| Software ontology | Kinds and structures of the objects, properties, and relations in software | To analyze structure and relationships between software elements, models, and meta-models |
| Software psychology | Human factors in computer systems | To study, model, and measure human behavior in creating or using software systems |

experience (anecdotal and experimental). Neither of these methods produces absolute truth, but rather a probable truth that depends upon human opinion or prejudice. *Software ontology* is another philosophy-oriented discipline that analyzes kinds and structures of the software objects, properties, and relations (the reader should not confuse software ontology with the ontologies, i.e., data models).

Postmodern computer languages, such as Perl, put the focus not so much onto the problem to be solved, but rather onto the person trying to solve the problem [67]. The way humans interact with software and the effect software structures and systems have on human behavior become a factor that influences software development methodologies and even the syntax of programming languages [61]. Such a human behavior in creating or using software systems is studied by *software psychology* [14, 23].

Software axiology analyzes the value and quality of software based on formal as well as aesthetic criteria (based on visual attractiveness of a software system or source code) [64]. The example of the latter can be literate programming [12], a problem which is totally irrelevant for computers, yet very important for programmers. *Software aesthetics* also has a practical level: often an elegant code runs faster, compiles better, and is more resource efficient and less prone to software bugs. Therefore, aesthetics and elegance in programming are often equivalent to good design [46].

Software archeology (or *software paleontology*) presents another view to software as a historical and material artifact and aims at the excavation of essential details about a software system sufficient to reason about it [3, 36, 34]. Archeology is a useful metaphor, because researchers try to understand what was in the minds of other software developers using only the artifacts they have left behind [41].

According to Minsky [48], computer programs are societies. The study of such societies and their relationship to human societies is a subject of *software sociology* or *social informatics* [26, 59]. Social

informatics envisions information systems in general and software in particular as a web-like arrangement of technological artifacts, people, social norms, practices, and rules. As a result, the technological artifact and the social context are inseparable [42]. All stages of software's life cycle are shaped by the social context [59]. *Software anthropology (ethnography)* uses field study techniques in industrial software settings to study the work practices of individuals and teams during SE activities, and their material artifacts such as the tools used and the products of those tools (documentation, source code, etc.) [35, 5, 66].

Large software systems follow the same distribution laws as social systems [1, 39]. OO programs have fractal-like properties [55] similar to natural, biologic, or social systems. These observations lead to software morphology, a discipline that studies forms or shapes of software, its parts (components, objects), and their representations. *Software gerontology* continues this view toward software as living entities, which grow, mature, and age [54]. *Software ecology* [43] becomes important, where sustainable software development and prevention of software pollution (i.e., harm to environments and users) are the main issues.

4. Socio-Technical Software Analysis

4.1. Concept, Context, and Aims of Socio-Technical Software Analysis

Analysis of a domain is the essential activity in SE, or more generally, in domain engineering. The aim of domain analysis (DA) is to recognize a domain by identifying its scope and boundaries, common and variable parts, which are then used to produce domain models at different levels of abstraction (such as feature models, UML models, source code). Though the objects of socio-technical analysis (STA) are artifacts of SE process in general, STA focuses on real world rather than on a particular domain problem. That is, the objects of study are the influence of used design methods, techniques, styles, programming practices, tool usage patterns, and the designer himself, his behavior, mental models, rationale and relationship with other designers, the organizational structure of a design team and business models on developed systems, and their quality and impact on other systems.

The new emerging discipline of STA should be viewed within the context of meta-engineering and meta-design. Meta-design [24, 15] extends the traditional system design beyond the development of a specific system to include the end-user-oriented design for change, modification, and reuse. A particular emphasis is given to increasing participation of users in system design process, and evolutionary development of systems during their use time when dealing with future uses and problems unanticipated at domain analysis and system design stages. A fundamental objective of meta-design is to create socio-technical environments that empower users to engage actively in the continuous development of systems rather than use of existing systems. Rather than presenting users with closed systems, meta-design provides them with opportunities, tools, and social structures to extend the system to fit their needs. Other approaches that argue for a more active user participation in IS and software design are ETHICS [49], soft system methodology [9].

STA includes the application of other empirical methods for studying complex socio-technical relationships between designers, software systems and their environment, including the social, organizational, psychological, and technological aspects. The ultimate aim of STA is the evaluation of design methodologies, the discovery of design principles, the formalization of mental models of designers, which precede design meta-models, comparison of design metrics, comparison of design subjects (actors, designers) rather than design objects (programs), discovery and analysis of design strategies, patterns and meta-patterns, and analysis of external factors that affect software design.

4.2. Socio-Technical Software Analysis vs. Traditional Domain Analysis

The following is a result of author's observations on differences between socio-technical analysis (STA) of software systems and domain analysis (DA).

In general, analysis is the procedure by which we break down an intellectual or substantial system into parts or components. DA is a part of SE that deals with the analysis of existing complex, large-scale software systems, and other relevant information in a domain (interactions within those systems, their development history, etc.) aiming to fill a gap in the business framework where a newly designed software system should exist. DA results in the development of domain models, which are further used for developing required software system(s).

The STA methods attempt to uncover information about software engineers by looking at their produced output (source code, comments, documentation, reports) and by-products (tool usage logs, program traces, events). It deals with the analysis of models and meta-models behind these systems and their application domain rooted in the mental models of system designers and social (organizational) structure of the environment. STA aims to understand complexity, interconnectedness, and wholeness of components of systems in specific relationship to each other.

DA focuses on separation and isolation of smaller constituent parts of a system, and analyzes their interaction and relationship. STA aims at expanding its view and including other related systems and domains in order to take into account larger number of interactions involved with an object of study. It adopts a holistic approach and focuses on the interaction of the study object with other objects and its environment, including other systems, domains, and the designer himself.

Traditional DA tends to involve linear cause and effect relationships. STA aims to include the whole complex of bidirectional relationships. Instead of analyzing a problem in terms of an input and an output, e.g., we look at the whole system of inputs, processes, outputs, feedback controls, and interaction with its environment. This larger picture can typically provide more useful results than traditional DA.

Traditional DA focuses on the behavior and functionality of designed domain systems (components, entities). The result is the data that characterize domain systems (e.g., its features, aspects, characteristics, and metrics). STA continues the DA further by analyzing data and content yielded during previous analysis stages using mathematical, statistical, and/or socio-technical methods. The aim is to obtain data about data (or meta-data) that help to reveal deeper properties of software systems that are usually buried in its source code or documentation.

STA does not replace the traditional DA methods, but rather extends them for deeper analysis and domain knowledge. The results of STA (*meta-knowledge*) can be used for increasing quality of software products, improving software design processes, providing recommendations for better management of design organizations, raising the level of education, spreading good design practices and programming styles, improving workplace conditions, etc.

5. Discussion and Conclusions

Software design processes and their artifacts have many perspectives: technological, social, cultural, and psychological. The psycho-socio-technical perspectives of software and IS development provide deeper insight into the relationship among methods, techniques, tools and their usage habits, software development environment and organizational structures, and allow to highlight the analytic distinction between how people work and the technologies they use. These perspectives can be traced to program source code analyzed and uncovered using social disciplines of computer science and the socio-technical software analysis methods.

The main object of research in the social disciplines of computer science is the virtual world of software and its relationship with the real world of designers and users. These disciplines reflect diverse views toward software as an environment (software ecology), imprints of the programmer's psyche (software psychology), artifacts of past information systems (software archeology), form of knowledge (software epistemology), growing and aging entities (software gerontology) that have their own form and shape (software morphology), and entities that interact with institutional and cultural contexts (software sociology).

Socio-technical analysis (STA) can be used for a number of problems, including program comprehension, plagiarism detection, design space exploration, and pattern mining. However, in practice it is very difficult to disentangle social aspects from purely technological aspects of software design, because they are mutually interdependent. The application of the STA methods may provide valuable insights into software development processes, the structure of the development team, the relationship of the software developers with their environments, understanding of programmer communication knowledge sharing, cognitive and mental processes of the developers and what influence it has on the quality and other characteristics of the produced software product. The results of STA can be used for improving programmer education, spreading good programming practices and styles, improving the management structure of the development team and the quality of its environment, and improving the performance of software design processes and quality of design artifacts (source code, documentation, etc.).

References

1. Adamic, L.A., Lukose, R.M., Puniyani, A.R. and Huberman, B.A. (2001) Search in Power-Law Networks. *Physical Review E*, 64, 046135.
2. Amrit C., Hillegersberg, J. and Kumar, K. (2004) A Social Network Perspective of Conway's Law. In *CSCW'04 Workshop on Social Networks*, Chicago.
3. Antón, A.I. and Potts, C. (2003) Functional Paleontology: The Evolution of User-Visible System Services. *IEEE Transactions on Software Engineering* 29(2), 151–166.
4. Bannon, L. (2001) Developing Software as a Human, Social and Organizational Activity. In *13th Workshop on Psychology of Programming (PPIG'2001)*, Bournemouth University, UK.
5. Beynon-Davies, P. (1997) Ethnography and Information Systems Development: Ethnography of, for and Within is Development. *Information and Software Technology* 39, 531–540.
6. Borchers, G. (2003) The Software Engineering Impacts of Cultural Factors on Multi-cultural Software Development Teams. In *Proc. of the 25th Int. Conf. on Software Engineering*, May 3–10, 2003, Portland, Oregon, USA, 540–547.
7. Bryant, S. (2004) XP: Taking the Psychology of Programming to the eXtreme. In *16th Workshop on Psychology of Programming (PPIG'2004)*, Carlow, Ireland.
8. Cannon-Bowers, J.E., Salas, E. and Converse, S. (1993) Shared Mental Models in Expert Team Decision-Making. In Castellan, J. (Ed.), *Individual and Group Decision-Making: Current Issues*. Lawrence Earlbaum and Associates, Inc., Mahwah, NJ, 221.
9. Checkland, P.B. (1999) *Soft Systems Methodology in Action*. John Wiley and Sons Ltd., New York
10. Chong, J., Plummer, R., Leifer, L., Klemmer, S.R., Eris, O. and Toye, G. (2005) Pair Programming: When and Why it Works. In *Proc. of Workshop on Psychology of Programming (PPIG 2005)*, University of Sussex, Brighton, UK.
11. Conway, M.E. (1968) How Do Committees Invent. *Datamation* 14(4), 28–31.
12. Cordes, D. and Brown, M. (1991) The Literate-Programming Paradigm. *Computer* 24(6), 52–61.
13. Curtis, B. (1988) The Impact of Individual Differences in Programmers. In van der Veer, G.C. (Ed.), *Working with Computers: Theory Versus Outcome*. Academic Press, New York, 279–294.
14. Curtis, B., Soloway, E.M., Brooks, R.E., Black, J.B., Ehrlich, K. and Ramsey, H.R. (1986) Software Psychology: The Need for an Inter-Disciplinary Program. In *Proceedings of the IEEE* 74(8), 1092–1106.
15. Damaševičius, R. (2006) On the Application of Meta-Design Techniques in Hardware Design Domain. *International Journal of Computer Science (IJCS)*, 1(1), 67–77.
16. Damaševičius, R. (2007) Analysis of Software Design Artifacts for Socio-Technical Aspects. *INFOCOMP Journal of Computer Science*, 6(4), 7–16.
17. Doherty, N.F. and King, M. (1998) The Importance of Organisational Issues in Systems Development. *Information Technology and People* 11(2), 104–123.
18. Douce, C. (2001) Long Term Comprehension of Software Systems: A Methodology for Study. In *13th Workshop on Psychology of Programming (PPIG'2001)*, Bournemouth, UK.
19. Douce, C. (2004) Metaphors We Program by. In *16th Annual Workshop on Psychology of Programming (PPIG'2004)*, Carlow, Ireland.
20. Eleutheros Manifesto. <http://www.eleutheros.it/documenti/Manifesto>
21. Eno, B. (1996) *A Year with Swollen Appendices*. Faber and Faber.
22. Espinosa, J., Slaughter, S. and Herbsleb, J. (2002) Shared Mental Models, Familiarity and Coordination: A Multi-Method Study of Distributed Software Teams. In *Proc. of 23rd Int. Conf. on Information Systems*, Barcelona, Spain, 425–433.
23. Finholt, T. (2004) Toward a Social Psychology of Software Engineering, Perspectives Workshop: Empirical Theory and the Science of Software Engineering, Dagstuhl Seminar 04051.
24. Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A.G. and Mehandjiev, N. (2004) Meta-Design: A Manifesto for End-User Development. *Communications of ACM* 47(9), 33–37.

25. Grinter, R.E. (2003) Recomposition: Coordinating a Web of Software Dependencies. *Computer Supported Cooperative Work* 12(3), 297–327. Springer.
26. Halavais, A. (2005) Social Informatics: Beyond Emergence. *Bulletin of the American Society for Information Science and Technology* 31(5), 13.
27. Hales, D. and Douce, C. (2002) Modelling Software Organisations, In Kuljis, J., Baldwin, L. and Scoble, R. (Eds.). *Proceedings of PPIG 2002*, Brunel University, UK, 140–149.
28. Hall, J.G. and Silva, A. (2003) A Requirements-Based Framework for the Analysis of Socio-Technical System Behaviour. In *Proc. of 9th Int. Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ03)*, 117–120.
29. Herbsleb, J.D. and Grinter, R.E. (1999) Splitting the Organization and Integrating the Code: Conway's Law Revisited. In *Proc. of Int. Conf. on Software Engineering*, Los Angeles, CA, 85–95.
30. Herbsleb, J.D. (1999) Metaphorical Representation in Collaborative Software Engineering. In *Proc. of Joint Conf. on Work Activities, Coordination and Collaboration*, San Francisco, CA, 117–125.
31. Hoc, J.M. and Nguyen-Xuan, A. (1990) Language Semantics, Mental Models and Analogy. In Hoc, J.M., Green, T.R.G., Samuray, R. and Gilmore D.J. (Eds.), *Psychology of Programming*, London: Academic Press, 139–156..
32. Holloway, C.M. (1995) Software Engineering and Epistemology. *Software Engineering Notes*, 20(2), 20–21.
33. Holt, R.C. (2002) Software Architecture as a Shared Mental Model. In *ASERC Workshop on Software Architecture*, Alberta, Canada.
34. Hsi, I., Potts, C. and Moore, M. (2003) Ontological Excavation: Unearthing the Core Concepts of the Application. In van Deursen, A., Stroulia, E. and Storey, M.-A. D. (Eds.), *Proc. of 10th Working Conference on Reverse Engineering (WCRE 2003)*, 13–16 November 2003, Victoria, Canada, 345–352.
35. Hughes, J.A., Somerville, I., Bentley, R. and Randall, D. (1993) Designing with Ethnography: Making Work Visible. *Interacting with Computers* 5(2), 239–253.
36. Hunt, A. and Thomas, D. (2002) Software Archaeology. *IEEE Software*, 19(2), 20–22.
37. Hvatum, L. and Kelly, A. (2005) What Do I Think About Conway's Law Now? *Conclusions of EuroPLoP2005 Focus Group*.
38. Iannacci, F. (2005) *The Social Epistemology of Open Source Software Development: the Linux Case Study*. PhD. Thesis, Department of Information Systems, London School of Economics and Political Science, London, UK.
39. Jing, L., Keqing, H., Yutao, M. and Rong, P. (2006) Scale Free in Software Metrics. In *Proc. of the 30th Annual Int. Conf. on Computer Software and Applications, COMPSAC 2006*, 1, 229–235.
40. Kersten, G.E., Kersten, M.A. and Rakowski, W.M. (2001) Application Software and Culture: Beyond the Surface of Software Interface. InterNeg Research Paper INR01/01.
41. Kerth, N.L. (2001) On Creating a Disciplined and Ethical Practice of Software Archeology. In *OOPSLA01 Workshop Software Archeology: Understanding Large Systems*, Tampa Bay, FL.
42. Kling, R., Rosenbaum, H. and Sawyer, S. (2005) *Understanding and Communicating Social Informatics: A Framework for Studying and Teaching the Human Contexts of Information and Communications Technologies*. Information Today, Inc.
43. Lanzara, F.G. and Morner, M. (2003) The Knowledge Ecology of Open-Source Software Projects. In *19th EGOS Colloquium*, Copenhagen, Denmark.
44. Letovsky, S. (1986) Cognitive Processes in Program Comprehension. In Soloway, E. and Iyengar, S. (ed.), *Empirical Studies of Programmers*, Ablex Publishing Company, New York, 58–79.
45. Luna-Reyes, L.F., Zhang, J., Gil-Garcia, J.R. and Cresswell, A.M. (2005) Information Systems Development as Emergent Socio-Technical Change: A Practice Approach. *European Journal of Information Systems* 14, 93–105.
46. MacLennan, B.J. (1997) Who Cares About Elegance? The Role of Aesthetics in Programming Language Design. *SIGPLAN Notices* 32(3): 33–37.
47. Marshall, L. and Webber, J. (2002) The Misplaced Comma: Programmers' Tales and Traditions. In *Proc. of PPIG 2002*, Brunel University, UK, 150–155.
48. Minsky, M. (1986) Introduction to LogoWorks. In Solomon, C., Minsky, M. and Harvey, B. (Eds.), *LogoWorks: Challenging Programs in Logo*. McGraw-Hill.
49. Mumford, E. (1995) *Effective Systems Design and Requirements Analysis: The ETHICS Approach to Computer Systems Design*. Macmillan, London.
50. Noble, J. and Biddle, R. (2004) Notes on Notes on Postmodern Programming. *SIGPLAN Notices* 39(12), 40–56.
51. Norman, D.A. (1986) Cognitive Engineering. In Norman, D.A. and Draper, S.W. (Eds.), *User Centred System Design*. LEA Associates, NJ.
52. Ørstavik, I.T.B. (2006) How Programming Language Can Shape Interaction and Culture. *Int. European Conf. on Computing and Philosophy, ECAP 2006*, June 22–24, Trondheim, Norway.
53. Parnas D.L. (1972) On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. of the ACM* 15(12), 1053–1058.
54. Parnas, D.L. (1994) Software aging. In *Proc. of the 16th Int. Conf. on Software Engineering*, Sorrento, Italy, 279–287.
55. Potanin, A., Noble, J., Frean, M. and Biddle, R. (2005) Scale-Free. Geometry in OO Programs. *Commun. of the ACM*, 48(5), 99–103.
56. Robles, G., González-Barahona, J.M. and Guervós, J.J.M. (2006) Beyond Source Code: The Importance of Other Artifacts in Software Development (A Case Study). *Journal of Systems and Software* 79(9), 1233–1248.
57. Rosen, C.C.H. (2005) The Influence of Intra-Team Relationships on the Systems Development Process: A Theoretical Framework of Intra-Group Dynamics. In *Proceedings of Workshop on Psychology of Programming (PPIG 17)*, Brighton, UK, 30–42.

58. Sawyer, S. and Guinan, P.J. (1988) Software Development: Processes and Performance. *IBM Systems Journal* 37(4), 553–569.
59. Sawyer, S. and Tyworth, M. (2006) Social Informatics: Principles, Theory and Practice. In *7th Int. Conf. 'Human Choice and Computers', IFIP-TC9 'Relationship between Computers and Society'*, Maribor, Slovenia.
60. Sharp, H.C., Woodman, M., Hovenden, F. and Robinson, H. (1999) The Role of Culture in Successful Software Process Improvement. In *Proc. of 25th Euromicro Conf.*, 8–10 September 1999, Milan, Italy, 170–176.
61. Shneiderman, B. (1980) *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop.
62. Souza, de C., Dourish, P., Redmiles, D., Quirk, S. and Trainer, E. (2004) From Technical Dependencies to Social Dependencies. In *Social Networks Workshop at CSCW Conf.*, Chicago, IL.
63. Souza, de C., Froehlich, J. and Dourish, P. (2005) Seeking the Source: Software Source Code as a Social and Technical Artifact. In *Proc. of Int. ACM SIGGROUP Conf. on Supporting Group Work, GROUP 2005*, Sanibel Island, FL, 197–206.
64. Tractinsky, N. (2004) Toward the Study of Aesthetics in Information Technology. In *Proc. of the Int. Conf. on Information Systems, ICIS 2004*, December 12–15, 2004, Washington, DC, USA, 771–780.
65. Trainer, E., Quirk, S., de Souza, C. and Redmiles, D.F. (2005) Bridging the Gap Between Technical and Social Dependencies with Ariadne. In *Proc. of the 2005 OOPSLA Workshop on Eclipse Technology eXchange (eTX)*, San Diego, CA, 26–30.
66. Viller, S. and Sommerville, I. (1999) Coherence: An Approach to Representing Ethnographic Analyses in Systems Design. *Human-Computer Interaction* 14(1,2), 9–41.
67. Wall, L. (1999) Perl, The First Postmodern Computer Language. Speech at *Linux World*.
68. Watt, S.N.K. (1998) Syntonicity and the Psychology of Programming. In *Proc. of 10th Workshop for the Psychology of Programming Interest Group (PPIG)*, Open University, UK, 75–86.
69. Weinberg, M.W. (1971) *The Psychology of Computer Programming*. Van Nostrand Reinhold, New York.
70. Zelkowitz, M.V., Wallace, D. and Binkley, D. (1998) Culture Conflicts in Software Engineering Technology Transfer. In *NASA Goddard Software Engineering Workshop*, Greenbelt, MD, USA, February 12, 1998.

Information Systems Development

Towards a Service Provision Society

Papadopoulos, G.A.; Wojtkowski, W.; Wojtkowski, G.;

Wrycza, S.; Zupancic, J. (Eds.)

2010, LIV, 974 p., Hardcover

ISBN: 978-0-387-84809-9