CODIFICATION AND TRANSFERABILITY OF IT KNOWLEDGE

Katarina Voutsina, Jannis Kallinikos and Carsten Sørensen
The Information Systems Group
Department of Management
London School of Economics and Political Science
Houghton Street, London WC2A 2AE, UK
{k.voutsina; j.kallinikos; c.sorensen}@lse.ac.uk

ABSTRACT

Software development has always been considered a complex undertaking where close interaction has been the antidote to this inherent complexity and development techniques from initial unstructured- over structured- and to object-oriented programming represent ways of managing development risks. Software knowledge has traditionally been transferred in project settings and been intrinsically linked with situated social practices. However, with the emergence of itinerant experts and highly distributed software development, the question emerges; what is the role of core software development techniques in the exchangeability and transferability of highly skilled IT knowledge? The aim of this paper is, through 30 qualitative interviews in Greece, to investigate the role of development techniques as a means of facilitating the codification and transferability of IT knowledge among itinerant IT experts and the projects they form part of. It is argued that the use of object-oriented techniques encapsulates discretionary decisions in objects and through carefully negotiated interfaces allows for the transfer and reuse across contexts. This minimises side effects and facilitates both the cultivation of complex middleware and the distribution of distinct work packages to individual itinerant experts.

Keywords: IT knowledge transferability, IT knowledge codification, Object-Oriented programming technique, contracting

Published as: Voutsina, K., J. Kallinikos, & C. Sørensen (2007): Codification and Transferability of IT Knowledge. In *Proceedings of 15th European Conference on Information Systems (ECIS),* St. Gallen, ed. R. Winter and H. Österle.

1 INTRODUCTION

In his famous selection of software development essays from 1975, Fredrick Brooks Jr. (1995 reprint) elegantly outlined the essentially challenges in large-scale software development in terms of the collective challenges imposed by software development projects. IT knowledge has since been conceptualised as residing in projects where intense interaction, negotiation, planning and management ensures the proper negotiation of mutual interdependencies in complex specifications (Humphrey, 1988). The transfer of IT knowledge can be viewed as a complex mixture of economic exchange and social control in terms of reliance on in-house organisational IT knowledge, of professional expertise of outsiders and of software packages (Scarbrough, 1995). This has traditionally implied strict limitations to the distribution and sub-division of IT work. However, over the last two decades there has been a continuous proliferation of subcontracting and consulting in knowledge-intensive sectors of the economy, such as the software industry or entertainment (Castells 2000; Matusik and Hill 1998; Laubacher and Malone 1997; Tilly and Tilly, 1998).

These developments provide evidence of a radical shift away from traditional practices of knowledge generation and manipulation and the assumptions underlying our understanding of these. Professional services, and in particular highly-skilled IT services, instead of being developed and provided in house are now procured in the market in accordance with all those rules underlying the exchange of goods or services (Barley and Kunda, 2004). The client-firm buys in the professional knowledge of the IT contractor and the IT contractor in turn sells accordingly their knowledge capital under the prescription of a well-defined, time-limited contract. IT contractors move from one client-firm to the other applying their specialized knowledge to solve diverse problems and thus serve as "itinerant" experts (Barley & Kunda, 2004). In this process significant IT knowledge in the form of modules and packages seem to be quickly adjustable and easily transferable to a diversified business population resembling the free exchange of off-the-shelf packages in an open market.

These developments suggest a shift in the status of IT knowledge. From being originally considered to be particularly complex and highly specific to the enterprise it was deployed (Goldthrope, 1998), IT knowledge nowadays seems to acquire the intrinsic characteristics of an exchangeable commodity, largely undifferentiated and primarily traded on the basis of price as opposed to quality or other unique characteristics. If this is the case then the traditional view of IT knowledge production and management is seriously challenged prompting the re-evaluation of the mechanisms by which IT knowledge is generated and diffused.

This paper inquires into the nature of IT knowledge and the forms by which it constituted and conceptually organized as a solid body of commercial services. The ultimate purpose is to explore the conditions explaining the *exchangeability and transferability of highly-skilled IT knowledge* across diverse organizational and institutional contexts. In particular, the paper aims at understanding the role of the core software development techniques on this transfer of IT knowledge. This is initially accomplished by a brief presentation of the prevalent programming methodologies that have governed the IS field from the mid seventies onwards. It comments upon the prospects of malleability and adjustability of IT knowledge that are associated with the development of IT methodologies. Key conceptual paths we pursue involve the exploration of the degree of reproducibility and manageability of the IT artifact. We suggest that the basic forms through which IT knowledge is currently developed epitomizes a drift towards standardization and recomposition where IT services are produced by combinations of already existing components, and this partly accounts for the marketization of IT services (Kallinikos, 2006).

The investigation is based on qualitative data gathered from thirty interviews with highly skilled IT professionals engaged in contingent forms of employment. Responders were asked about their conceptualizations, the shifts in software programming techniques and how these shifts are linked to contemporary phenomena of freelancing and knowledge commodification. The empirical data suggests that the shift from *unstructured-* to *procedural-* and *object-oriented programming*, implies novel ways of managing the ambiguity and complexity inherent in any software development process promoting knowledge manoeuvreability and applicability across contexts.

The following section outlines the evolution of the IT development process. Section 3 presents the empirical study. Section 4 presents and discusses the results and Section 5 concludes the paper.

2 THE EVOLUTION OF INFORMATION TECHNOLOGY DEVELOPMENT PROCESS

The software development literature suggests three broad approaches in software development: unstructured programming also known as the code-and-fix approach, and two modular programming approaches; procedural programming and object-oriented programming (Boehm, 1988). The shift in the 1970s towards procedural or structured programming was touted as a "paradigm shift" in the IS industry. Years later, advocates of the object-oriented analysis herald the new "paradigm shift" object programming represented (Gibbs, 1994). In both cases, advocates of each approach aimed to draw the attention to the fact that the new approach of building software is not just a refinement or an updated version of the previous one, but a radical departure from a well-established way of thinking about software development. The three approaches on software development represent alternative strategies or sets of techniques that have been deployed by developers over time in an attempt to build more reliable and efficient software. The underlying logic and the challenge behind every engineering project might be summarized along the following issues (Brooks 1995):

- How to combine programs understood as an organized list of instructions into a system
- How to design and build a system into a robust, tested, well-documented and supported product that will process info in order to perform a specific task
- How to maintain intellectual control over the complexity in large doses.

Despite the commonality of the confronted problems, the way each software development approach classifies and manipulates information is strikingly different. As will be illustrated in the rest of the paper, each approach also hosts different possibilities with respect to knowledge organization and diffusion across contexts. The following briefly describes key aspects of each of the three approaches and subsequently these distinctive characteristics will be related to the codification and transferability of IT knowledge.

Unstructured programming essentially implies that all code is written as a single continuous block of instructions that are difficult even for the same programmer to separate, detach or even distinguish.

Structured programming in many respects relies on similar basic conceptual assumptions as those underlying *Scientific Management* (Taylor, 1911). The ultimate aim has been the detailed functional or logistical description and decomposition of tasks and operations, by focusing on the way information (data) is flowing and processed step-by-step (DeMarco, 1978; Yourdon, 1989). "*The analysis aims at rationalizing information processing by identifying and removing the procedural, historical, political or tool related peculiarities*" and identifying sequence and succession of functions (Bansler and Bødker, 1993).

Object Oriented (OO) programming rejects the decomposition of a system into processes and data flows, and instead draws attention to the decomposition of a system into independent interacting objects. Each object encapsulates both data expressed as variables and attributes, and processes in terms of functions, behaviors, and methods (Dahl & Nygaard, 2002). An object's behavior is enacted or an object's method is invoked when it receives messages or calls from another object. Through the principle of information hiding (Parnas, 1972), objects encapsulates discretionary and transient decisions and ensures minimal side-effects through the maintenance of carefully negotiated and stable interfaces.

Although there has been a burgeoning body of research that aims to compare and disclose the conceptual and practical differences of the above methodologies, the potential superiority or inferiority of these methodologies seems to remain largely elusive and the deriving findings are often accompanied by conflicting results (Johnson, 2002). The aim of this paper is not to present the benefits and drawbacks related to the deployment of each methodology as such, but rather to explore how their inherent characteristics favor or impede IT knowledge transferability and adaptability across different organizational contexts.

3 EMPIRICAL STUDY

The analysis that is presented in the following sections is based on data gathered through thirty interviews with IT professionals working as independent contractors, or itinerant experts, in Greece. This group has been deliberately selected as the working practices epitomize knowledge packaging and transfer across different organizational and business contexts. Eight out of the thirty interviewees, were general IT consultants and managers. Five of the interviewees had highly specialized skills in a very particular technology or commercial off-the-shelf software package such as those manufactured by SAP (www.sap.com). The remaining 17 interviewees were specialized in a wider range of technologies. All of the interviewees had university degrees in computer science or related subjects and all had at least five years work experience. (Table 1 in the "appendix section" highlights some of the main interviewee characteristics).The interviews were conducted in the period between september 2005 and june 2006 and each of them lasted from one hour and a half to two hours.

The respondents were asked to describe how they conduct their work, reflecting upon the conceptual tools and methodologies used, as well as the functionalities the latter entail in allowing them to move freely from across organizational boundaries and contexts. They were then asked to comment upon and compare alternative programming methodologies in terms of the degree of IT artifact reproducibility and maneuverability.

Given that there is no an established classification of IT individuals who work as free-lancers, the selection of informants was not a straight-forward process. Informants were selected from a list of the Federation of Greek IS enterprises and IS personnel, following the logic of a snowball sampling, i.e., respondents were asked to provide details of others they deemed interesting for the study (Evans et al. 2004; Faugier and Sargeant, 1997;). Even if our respondents cannot be considered as representative of the relevant population in Greece, and therefore prohibiting us from arriving at statistical generalization, their explanations and testimonies contribute to what Yin (2003) characterizes as analytic generalization, i.e. they promote our overall understanding with respect to the relation between software methodologies, knowledge codification and knowledge transferability.

The in-depth semi-structured "ethnographic" interviews lasted one to one and a half hours and were tape-recorded at the participants' work places in Athens (Barley et al, 2002). Furthermore, some informants were interviewed twice, because the initial transcription of their sayings rendered necessary a second round of interview in order to clarify issues that remained unclear.

4 IT KNOWLEDGE CODIFICATION AND ENABLERS OF EXCHANGEABILITY

The current section rely upon the interview narratives to obtain an understanding of the defining differences between the prevalent methodologies of software development (unstructured, procedural and object-oriented) and how these are associated with the freelancing practices of the interviewed IT professionals.

For all interviewed professionals it is common sense unstructured design is directly associated with greater zones of ambiguity and vagueness in comparison to any other kind of structured programming. Although any kind of software language and programming is cognitively organized around explicit rule-based combinations of standardized binary code of 0 and 1, the reproducibility and controllability of unstructured "GOTO" programming are significantly lower in comparison to modular programming. Alternatively, the tacit knowledge needed to make sense and use of codified but still unstructured programming is significantly broader than that associated with modular programming. Granted that complexity is an inherent property of the software produced and that the different components unavoidably interact with each other in a non-linear fashion (Brooks, 1987), it is plausible to assume that managing and maintaining a single and continuous block of code is particularly difficult.

Against this background it would seem reasonable to argue that the quality of the final result is highly contingent upon the idiosyncratic skills of the highly trained craftsman and strongly prescriptive for the particular enterprise and the particular tasks it is brought to bear upon. No matter how much the final result can be judged as satisfactory and the process by which it is reproduced can be documented, the actual cognitive trajectory followed by the developer remains at least at some extend vaguely defined and not easily communicable and shared.

"*Unstructured programming is a real art. If the programmer who writes the code is really gifted in his/her craft, the deriving result could be described as a masterpiece. Yet, maintenance of such a program is a major challenge for anyone other than the initial manufacturer of the program and its transferability across different enterprises is rather absent. We could never work as contractors if we had remained faithful to the principles of unstructured programming*". (developer no.27, table 1, appendix)

The antidote to the aforementioned limitations of unstructured programming could be no other than the breaking of programs/systems into smaller components that can be constructed independently of one another and then be recombined to make the overall system: the rational behind the *modular programming*. Here, principles of low inter-module coupling and strong intra-module cohesion have been applied as quality criteria to strive for (Sommerville, 1982).

Structured or procedural methods and object-oriented (OO) methods both aim at reducing complexity and enhancing visibility by creating various kinds of classifications, functions, sub-functions and objects. The process of classification is itself a method of standardization since it ends up imposing a degree of homogenization that renders the particular individual differences irrelevant and rather insignificant (Bowker and Star, 1999; Townley, 1994). Deciding upon what is important and what is not, thinking to which extend something is similar or dissimilar to something else, is the first step of rendering oneself able to understand and manipulate a set of relations that otherwise would have remained closely interwoven and practically non-exploitable. Here, the process is guided by the principles of maintaining a low degree of inter-module coupling and a high degree of intra-module cohesion ensuring that each module is a coherent unit. Parnas' (1972) conceptualized and strengthened this principle in terms of information hiding where as a design principle, the internals of a module should exercise information hiding and encapsulate discretionary modeling decisions and subsequently publish clearly negotiated and stable interfaces for other modules to access. Object oriented principles for describing a system in modules is a very effective means of implementing these principles and furthermore offers additional principles such as polymorphism and enheritance.

Most interviewees mentioned that modular programming and the functionality embedded in it constitutes the fundamental pillar of IT contracting. At the beginning of the interview they talked about both processes of building software as being equally useful and usable, "*Although each IT solution generated is highly specific and tailored-made to the needs of a particular client, existing lines of code are always being re-used in our craft, independently of whether this is a subroutine, a function or an object"* (developer no. 30, table 1, appendix). However, as the discussion proceeded they tended to display a kind of indirect and semi-articulated preference towards object-oriented methodologies.

What are then the differences of codification and cognitive organization, standardization, between structured and object oriented approaches? Drawing upon the differences between their subordination to different levels of standardization, there is a clear distinction between the way the programmer conceptualizes and makes sense of the program, and the way in which the program operates to execute the predefined, codified instructions underlying the performance of a specific task.

As far as the execution of a software program is concerned, every single sequence of instructions is translated into a unique combination of binary coding and consequently it is syntactically and semantically differentiated. Every software program operates under the assumption of "*frozen signification, fixed one-to-one correspondence and clear-cut and finitely differentiated semantic units*" (Kallinikos, 1996). Two different syntactic units can never refer to the same semantic unit. Whatever is presented as semantic content is not but another syntactic notation that seeks to fix or agree upon the content it describes (Simon, 1977). Therefore, the software program independently of the process of its products, is the result of a highly codified process. Nevertheless, there is a great difference between the procedural and the OO approach of software development in the way the developer manages the ambiguity related to the management of a particular function or of an object.

4.1 Capturing and encapsulating ambiguity

In procedural programming the data is separated from the procedures and are global, easily reachable and appropriated by different equations and functions. The programmer has to describe the function and specify the types of variables to be used as the specific function to deliver the desired outcome. Given the fact that several functions have access to global data, a function can modify data that are outside its scope or contribute to the corruption of data prospectively used by other functions. This phenomenon is characterized as a "side-effect" and makes the behavior more difficult to predict. The overall quality of the generated solution is contingent upon the depth of knowledge the programmer possesses regarding programming techniques and the overall structural formula, coupling and coherence of the system. It is up to the programmer to decide what kind of function has to be developed and what type of data and input values need to be used; and yet even if he or she performs everything by the book, the final outcome still remains significantly vague. The inherent complexity and incontrollable interactions and interdependencies of diverse components that share the same pool of data render the prediction of the final result rather problematic.

On the other hand, in object-oriented programming, the process followed by the programmer to build a system is at least at a certain degree more standardized and the expected outcome is relatively more predictable. The data (attributes) and procedures (methods) are encapsulated within a single independent entity, the "object", whose internal structure remains a black-box to the other components of the system. The object can only be accessed via its external behavior (methods), while its attributes (data) are meticulously hidden and carefully protected through the application of information hiding. Concomitantly, the programmer does not really have to worry about how she will make the "object" to function in a particular way. She has just to call, to invoke the behaviors of an object and ask from it to perform an operation, most often on itself, specific to itself, using its own data. More precisely, objects communicate with each other by exchanging messages, which constitutes a second order codification of an already consolidated knowledge. This knowledge is embedded within the object itself and is represented by the correspondence between methods (behaviors) and attributes (data).

Each abstract class inherits its methods and attributes to its sub-classes, and each sub-class is differentiated from its super-class (parent class) by displaying some additional methods and attributes. What is really remarkable is that the same message will be processed differently from diverse subclasses of the same class according to their unique identity (polymorphism). Alternatively, the methods of each subclass will be separately enacted in the specific "context" of the particular object. Therefore two objects that display exactly the same output to a predefined message cannot but be the same object. In terms of semantics, the "meaning" of each object response is unique and univocal. In contrast to the case of procedural programming where the "meaning" of the outcome may vary according to diverse contingencies, related to interdependencies, unplanned interactions and possible data corruption.

In other words, a part of what has previously been tacit knowledge associated with which functions should be linked to what particular types of data and which specific data the latter should choose to deploy is already codified and embedded into the very notion of the object itself. In other words, the final product is substantially separated from the process by which it is constructed. The individual developers discretionary choices beyond those carefully negotiating interfaces between objects are effectively hidden and protected against side-effects. Through the development of object programming the process of developing software becomes relatively independent from the skills and proclivities of individual programmers (Goodman, 1976; Kallinikos, 2002). Knowledge development is standardized. The way object-oriented programming methodologies are conceptually organized subtantially contributes to increasing the degree of standardization both in the way code is written (objects) and the relationships between different strings of codes (classes inheritance) are established.

"*Software programs built upon the logic of object-oriented programming are much easier to communicate and understand, since there is a specific degree of accountability to the diverse components of the system-objects. Each object has a specific level of responsibility and if something goes wrong, one does not have to worry about tracking down every line of code that might have changed a specific attribute-instead he/she has to look into the very structure of the object that controls the specific attribute. Information hiding constitutes a unique way to reduce complexity and enhance clarity of interactions"* (developer no.19, table1, appendix).

The interactions of the component parts of the object oriented system are much less ambiguous and more stable and visible than those found in the procedural programming. Ambiguity is encapsulated in the internal structure of the object, and this way is significantly diminished, since the complexity underlying the internal structure of the object is considerably smaller to the complexity found in large systems with many interacting components.

4.2 Possibilities of adaptability and transference

Increased standardization and visibility of the software building process brought about by the object-oriented programming cannot but path the way towards enhanced possibilities of IT knowledge transferability across different organizational environments. Kallinikos and Hasseldbladh (2000), in an attempt to distinguish those characteristics that allow some ideas and managerial practices to diffuse across different contexts in a greater range than others, argue that the cognitive organization of such rationalized packages is key to their diffusion pattern. They, further, forward three central qualities describing the cognitive organization of such packages:

- the easiness/difficulty by which such packages can be locally reproduced,
- the degree of durability/perishability that they exhibit as they move across contexts
- the immediacy of communicability and the comprehensibility by which local actors encounter such artifacts/packages.

Reproducibility is here a key aspect to the understanding of subcontracting and freelancing practices in the IT industry. By separating the product (IT services) from the process by which it is constructed, object oriented approaches considerably raise the reproducibility of IT artifacts and contribute to their transferability across contexts

The analytic paths undertaken so far therefore demonstrate that maintaining and reproducing a single and continuous block of code (the product of unstructured programming methodologies) is so difficult, even for the person who has initially created the program, that the prospect of transferring and applying parts of this code to a new application seems to be completely absent or at any rate very low. The distinction between procedural programming and object-oriented programming in terms of the possibilities of knowledge transferability that each methodology accommodates is more difficult to establish. The key concept that helps to unravel the difference between the two programming methodologies resides in the notion of inheritance (inheritance of methods and attributes) and independency (independency of diverse components) that the latter embraces.

Different independent components (objects) of a system can be easily combined and separated, allowing the system to be easily decomposed and composed anew according to the imperatives dictated by the business objectives. The more standardized the combinatorial rules of a system are the easier its reproducibility and adjustability are. In other words, standardizing the relationship between the independent components of the system is the first step to render it malleable and controllable and this is exactly the case of object oriented programming.

While procedural programming also provides code re-use by allowing the programmer to create a function (a procedure) and then use it again in multiple projects and for multiple purposes, object oriented programming goes one step further by allowing the programmer to define abstract but simultaneously straightforward "relationships" between independent classes of objects.

As already mentioned above, the concept of inheritance allows the developer to create brand new classes by abstracting out common attributes and behaviors. In particular, each abstract class inherits its methods and attributes to its sub-classes enabling the programmer to write the code for them just once. In other words, whenever the developer creates a new sub-class or instantiates a class into a new object, he or she has just to write only the new methods or attributes since all the other methods and attributes are automatically inherited by the parent class. Therefore, independently of the on-going application, an already part of the code is already ready-made waiting to accommodate new usages.

When the implementation has to change to meet the particular needs of a particular client firm, the developer does not have to worry about changing the interface. Different client-firms are receiving different software applications with the same functionality and the same interface. What changes is the intrinsic structure of an object that addresses the concerns of the particular firm or context to which the system is called to bear upon.

An object can be transferred from one application to another without significant semantic alterations or distortions as it signifies a self-sufficient whole of a public interface for sending and receiving messages hiding the idiosyncratically designed intestines. Performing an operation, most often on itself, specific to itself, and by using its own data, the object more or less delivers quite standardized and relatively predefined outputs. On the contrary the final output of the function, when transferred to a new context, may be non-uniform, because of the emerging interactivity patterns of the latter with other functions and the subsequent sharing of common data.

Finally, as far as communicability is concerned, most of the interviewed professionals reassured us that it was easier to work with other contractors and achieve better overall integration of the system under construction. This was the outcome of the fact that the standards of the common interfaces that objects would interact among themselves could be communicated better and quicker than the details related to the intrinsic structure and objective of diverse function interdependent functions.

To conclude, we would like to draw the attention to the fact that the aforementioned remarks do not necessarily imply downplaying the importance of tacit knowledge needed in the use and manipulation of this codified and relatively standardized knowledge. *"Codification is never complete and some forms of tacit knowledge will always play an important role"* (Cowan and Foray, 1997). In particular, the ability that developers might display in deeply comprehending the business objectives and translating them into their computational vocabulary resides into the field of tacit knowledge and constitutes one of the most significant factors that determine the final success of the project. (Curtis et al. 1988). Yet, the intention of the current section has been to demonstrate that the relative degrees of codification and standardization underlying the different software methodologies are instrumental in rendering the packaging of IT knowledge increasingly feasible.

5 CONCLUSIONS

Aim of the current paper has been an attempt to identify the structural features and intrinsic qualities of software development techniques that justify or partial explain the transferability of IT knowledge across different organizational contexts. Empirical data from thirty highly-skilled IT contractors in Greece suggest that Object-oriented techniques and the way the latter are conceptually structured and welcoming the human agency seem to lie in the heart of IT knowledge transferability and IS contracting.

As we move from unstructured to procedural and then to Object-oriented programming, the way software is built tends to lose its tacitness and heterogeneity and starts acquiring the form of a well-structured and uniform activity governed by explicit rules and procedures. An increasingly significant part of the individual developers' discretionary choices seems to be captured by the logic of more and more refined classifications and standardized correlations that tie the structural components of a system together. Visibility, clarity and understandability of the software production process are enhanced and the prospects of adjustability and malleability of the ICT artifact are more than enough. To put it in other words, IT knowledge seems to be partially imprisoned within the ICT artifact and gradually detached from its initiator, production process and context of application; acquiring the characteristics of a traded commodity that is freely exchanged and transferred across organizational settings, IT knowledge cognitive production is witnessed to become one of the fundamental enablers of contingent forms of IT work organization.

6 REFERENCES

Bansler and Bødker, (1993), "A Reappraisal of Structured Analysis: Design in an Organizational Context", *ACM Transactions on Information Systems*, 11, 2, 165-193

Barley, S.R. and G. Kunda, (2001), "Bringing Work Back In". *Organization Science*, 12, 1, 76-95

Barley, S.R. and G. Kunda, (2004). *Gurus, Hired Guns, and Warm Bodies: Itinerant Experts in a Knowledge Economy* Princeton University Press.

Boehm, B. W. 1988. "A Spiral Model of Software Development and Enhancement". *IEEE Computer,* 21 (5).

Booch, G, (1994), Object-Oriented Analysis and design with applications, 2nd edition Benjamin/Cummings (Redwood City, CA)

Bowker G. & Star S.L. (1999), Sorting Things Out: Classification and its consequences, Cambridge, MA, The MIT Press

Brooks, F., (1987), "No Silver Bullet: Essence and Accidents of Software Engineering", Computer, 20,4, 10-19

Brooks, F, (1995), The mythical man-month, 20 year anniversary edition. Addison-Wesley Longman Inc.

Castells M., (2000), "Materials for exploratory theory of network society", British Journal of Sociology, 51, 1,5-24

Cowan R., and Foray D, (1997), "The Economics of Codification and the Diffusion of Knowledge", Industrial and Corporate Change, 6, 3, 595-622

Curtis B., Krasner H. and Iscoe N. (1988), "A field study of the software design process for large systems", *Computer Practices*, 31, 11, 1268-1287

Dahl, Ole-Johan, and Kristen Nygaard. 2002. How Object-Oriented Programming Started. http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html.

DeMarco T., (1978), *Structured Analysis and systems specification*, Prentice-Hall, Englewood Cliffs, NJ

Evans A.J., Kunda G. and Barley R.S., (2004), "Beach Time, Bridge Time and Billable Hours: The Temporal Structure of Technical Contracting", *ASQ*, 49, 1-38

Faugier J., and Sargeant M., (1997), "Sampling harfd to reach populations", *Journal of Advanced Nursing*, 26, 790-797

Gibbs W.W. (1994), "Software's chronic crisis", *Scientific American*, 271, 3, 86-95

Goldthrope J.H. (1998), *On Sociology: Numbers, narratives and the integration of research and theory*, Oxford: Oxford University Press

Goodman N., (1976), *Languages of Art*, Indianapolis: Hackett

Humphrey, Watts S. 1988. "Characterizing the Software Process: A Maturity Framework". *IEEE Transaction of Software*

Engineering 5 (2):73-79.

Johnson A. R., (2002), "Object-Oriented Systems Development: A review of empirical research", *Communications of the Association for Information Systems*, 8, 65-81

Kallinikos J., (2002), "Reopening the black box of technology, artifacts and human agency", *23d International Conference on Information Systems*

Kallinikos J., (2006), The consequences of information: Institutional implications of technical change, Edward Elgar, Cheltenham

Kallinikos J. and Hasseldblah H., (2000), "The Project of Rationalization: Actirique and Reappraisal of Neo-Institutionalism in Organization Studies", *Organization Studies*, 21, 4, 697-720

Laubacher and Malone 1997 Laubacher R.J. and Malone T.W. (1997), Flexible work arrangements and the 21th century workers' guilds, MIT Sloan School of Management Initiative on Inventing the Organizations of the 21th Century, Working papers No.004

Matusik S.F. and Hill C.W.L. (1998), The utilization of contingent work, knowledge creation and competitive advantage, Academy of Management Review, 23/4: 680-697

Parnas, D. L. (1972). "On The Criteria To Be Used in Decomposing Systems into Modules". *Communications of the ACM,* 15 (12):1053-1058.

Scarbrough, Harry. 1995. "Blackboxes, Hostages and Prisoners". *Organization Studies* 16 (6):991-1019.

Simon, H., (1977), *The New Science of Management of Management Decision,* Englewood Cliffs: Prentice Hall

Sommerville, Ian. 1999. *Software Engineering*. Addison-Wesley.

Taylor F., W., (1911), *Scientific Management*, NY and London

Tilly C. and Tilly C., (1998), Work under capitalism, Oxford: Westview Press

Townley B., (1994), *Reframing Human Resource Management*, SAGE Publications Ltd.

Yin K.R., (2003), *Case Study Research: Design and Methods*, Sage Publications, Inc.

Yourdon E., (1989), *Modern Structured Analysis*, Yourdon Press/Prentice Hall, New York

7 APPENDIX

Table 1: Interviewees' technical specialities, projects involved into and years of experience

No.	*Technical Specialty*	*Projects*	*Years of experience*
1	IT specialist	SAP customized applications	6
2	IT specialist	SAP customized applications	8
3	IT specialist	SAP customized applications	7
4	IT specialist	SAP customized applications	5
5	IT specialist	SAP customized applications	7
6	IT consultant	Management of IT projects	10
7	IT consultant	Management of IT projects	20
8	IT consultant	Management of IT projects	9
9	IT consultant	Management of IT projects	12
10	IT consultant	Management of IT projects	15
11	IT consultant	Management of IT projects	12
12	IT consultant	Management of IT projects	15
13	IT consultant	Management of IT projects	17
14	Web developer	Development and maintainance of websites	5
15	Web developer	Development and maintainance of websites	5
16	Web developer	Development and maintainance of websites	5
17	Web developer	Development and maintainance of websites	7
18	Web designer	Development and maintainance of websites	8
19	Web designer	Development and maintainance of websites	10
20	Database designer	Databases creation and management	6
21	Database designer	Databases creation and management	8
22	Architecture designer	Software designing	12
23	Software developer	CRM applications	10
24	Software developer	Network applications	9
25	Software developer	Network applications	17
26	Software developer	Supply chain Applications	16
27	Software developer	ERP applications	12
28	Software developer	ERP applications	14
29	Software developer	ERP applications	14
30	Software developer	ERP applications	16