Analysis of Software Design Artifacts for Socio-Technical Aspects

Robertas Damaševičius
Software Engineering Department,
Kaunas University of Technology
Studentų 50-415, 51368 Kaunas, Lithuania
robertas.damasevicius@ktu.lt

Abstract. Software systems are not purely technical objects. They are designed, constructed and used by people. Therefore, software design process is not purely a technical task, but a socio-technical process embedded within organizational and social structures. These social structures influence and govern their work behavior and final work products such as program source code and documentation. This paper discusses the organizational, social and psychological aspects of software design; and formulates the context, aims, principles, and techniques of socio-technical software analysis. An illustrative case study demonstrates the application of the socio-technical software analysis method for estimating the extent of code sharing within programmer groups using the proposed Social information sharing metric.

Keywords: socio-technical software analysis, program comprehension, sociology of programs, code sharing.

(Received December 08, 2006 / Accepted February 28, 2007)

1 Introduction

1.1 Factors influencing software development

Software engineering is primarily concerned with developing software systems that satisfy functional and non-functional requirements, internal and external constraints as well as other requirements for modularity, comprehensibility, reusability, maintenance, documentation, etc. Such requirements and the developed software systems reflect organizational and social expectations of how, where, when, and why the software system may be used.

Software developers are not influenced just by the presented requirements and constrains. 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, organizational structure, social relations, shared mental models [15], and even such minor aspects whether a coffee machine is installed in their workplace [22].

Therefore, software design process is not purely a technical task, but also a social process embedded within organizational and cultural structures [22]. Software programmers collaborate in teams and groups embedded within larger organizations. These social structures influence and govern their work behavior and final work products such as program source code and documentation.

1.2 Socio-technical view on software design

Software systems are not purely technical objects. They are designed, constructed and used by people. Therefore, they are components in larger socio-technical systems which include technological as well as social structures. Therefore, social and cognitive issues should be addressed in designing and analyzing software.

These socio-technical relationships are very complex to register and study, and they cannot be replicated experimentally or using formal models. The actions and environment of software designers is rarely directly available for study. Often the only available material for analysis is the results of the programmers' work such as program source code. It can tell us about software design processes, its development history, and provide us with some information about its author. Comprehension of source code may allow us to

comprehend what the original programmer had comprehended [13].

From the socio-technical perspective, the structure of software systems can be described in terms of technical relationships between software elements (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.

1.3 Scope and aims of the paper

Socio-technical software analysis [11, 12, 23], tries to uncover these socio-technical dependencies by analyzing artifacts of software design processes. Socio-technical software analysis is a process of studying complex socio-technical relationships between software designers, software systems and their environment.

The aims of the paper is to consider the socio-technical aspects of software design reflected in program source code and to discuss the socio-technical software analysis methods for discovering social, organizational and psychological aspects embedded within it.

2 Social, organizational and psychological aspects of software design

2.1 Social aspects

System development is a socio-technological process [40]. It has long been recognized that personal [8], and group [46] factors affect systems development process. Sawyer and Guinan [41] even claim that social processes had more influence on software quality than design methodologies or automation. Unless human factors are taken into account, in particular, interpersonal relationships that affect the operation of the process, a vital component is being overlooked.

There are numerous evidences that software design processes are influenced by social and psychological factors [2, 4, 6, 14, 22, 24, 35, 40, 44]. Software engineers often express dependencies between code modules as social dependencies between people and groups, and that dealing with code integration is an organizational rather than purely technical matter [42]. The social nature of software development and use suggests the applicability of social psychology to understanding aspects of software engineering.

Programmers do not exist in isolation. They usually communicate about technical aspects of their work. Recent *ethnographic* studies [21, 11, 43] suggest that technical dependencies among software components create "social dependencies" among software developers implementing these components. For example, when developers are working to implement software system within the same team, the developers responsible for developing each part of the system need to interact and coordinate in order to guarantee the smooth flow of work [43]. Another example is when the developer implements the same or similar task and uses the results (with or without modifications) of other developer's work, who may be aware or unaware of this fact. 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 [12].

Therefore, software development (certainly at a large-scale) can be considered as a fundamentally social process embedded within organizational and cultural structures. These social structures enable, constrain and shape the behavior, knowledge and general programming techniques and styles of software developers [22].

2.2 Organizational aspects

Another kind of socio-technical aspects that influence the work of software designer are organizational aspects, which comprise the structure of organization, management strategy and business model. Such dependence is often formulated as *Conway's Law*. It was formulated by M. Conway: "*organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations*" [7].

There are numerous interpretations of Conway's Law [1, 25]. However, in general, Conway's Law states 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 (see Figure 1).

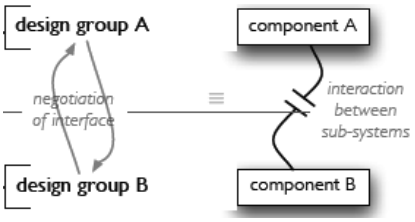

Figure 1: Relationship between design groups and designed components of a system

D.L. 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*" [38], which mean 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 [25].

The analysis of software architectures allows us to make conclusions about the organizational structure and social climate of the designer team. Therefore, socio-technical software analysis can be used to uncover information about organization from the software artifacts and documentation produced by it.

2.3 Psychological aspects

Software design decisions are often based on *psychological rationale*, rather than purely computational or physical factors. 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 [44]. About 70% of software representations are metaphorical, representing system behavior as physical movement of objects, as perceptual processes, or anthropomorphically by ascribing beliefs and desires to the system [24].

Software architecture is commonly considered to be the structure of a software system. However, software architecture also can be analyzed as a *mental model* shared among software developers [5, 27, 37]. 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, organization and psychological aspects that influenced the development of this system [26]. Mental models enable designers to acquire conceptual and causal networks and their associated processes, and facilitate their ability to generalize, conceptualize and interpret the outcomes of design [36].

Understanding mental models involved in programming provide the basis for improving software design, evolution and maintenance processes and designing higher-quality tools. Uncovering and analyzing a mental model of a program is as important as analyzing formal or abstract models, and contributes towards a more comprehensive understanding of the software and systems development processes.

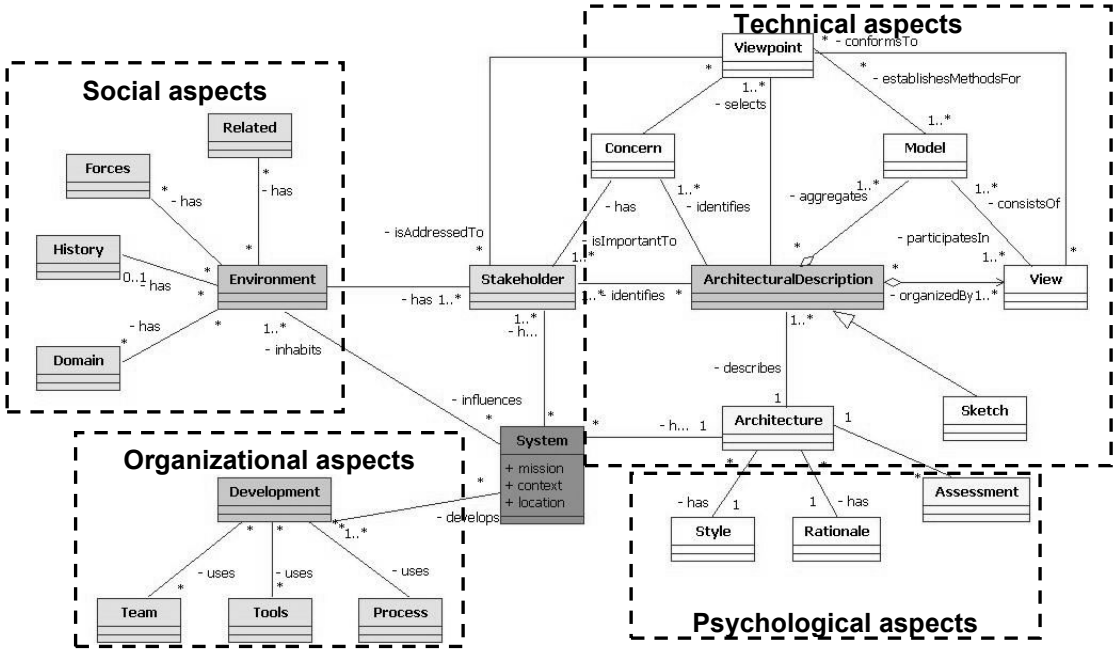

Figure 2: Software architecture meta-model (according to [3])

2.4 Socio-technical aspects and software architecture meta-model

The organizational, sociological and psychological aspects of software design can be directly extracted from software architecture meta-model [3] (see Figure 2).

Here, the sociological components of the meta-model are *Forces* of *Environment, History* and *Related Environment*. The organizational component of the meta-model is *Development*, which includes *Team*, available design *Tools* and applied development *Processes*. The psychological components of the meta-model are *Rationale* and *Assessment* of the *Architecture*. *Style* is also influenced by designer psychology. The remaining parts of the software architecture model relate to the technical and application-oriented aspects of software design.

3 Socio-technical software analysis

3.1 Concept of socio-technical analysis

Analysis of a domain is the essential activity in software engineering, or more generally, in *domain engineering* [19]. The aim of domain analysis is to recognize a domain by identifying its scope and boundaries, common and variable parts, which are then used to produce domain models. Domain models at different level of abstraction (such as feature tables, generic architectures, requirement models, UML models, source code) are the output of domain analysis.

However, the domain models themselves can be the object of higher-level analysis, or socio-technical software analysis. Whereas the aim of analysis is the creation of domain models, the aim of socio-technical software analysis is the creation of domain *meta-models* (such as statistical models, mental models, sociograms [43], etc.) that reflect the relationship between software, its designer and their environment.

Though the objects of socio-technical analysis are artifacts of software engineering process in general, the aims of socio-technical analysis focus 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 software systems, their quality and impact on other software systems. Thus, socio-technical analysis is strongly related with such fields as sociology, cognitive psychology [31] and human-computer interaction.

While traditional domain analysis methods are concerned with "*domain archeology*", i.e. extraction, classification of knowledge from the domain of study, socio-technical software analysis can be called a "*software archeology*" discipline. Software archeology is the recovery of essential details about an existing software system sufficient to reason about it [3, 39]. Archeology is a useful metaphor, because programmers try to understand what was in the minds of other software developers using only the artifacts they have left behind with the goal of not just understanding the artifact, but through the artifact we come to understand human life and culture [30].

3.2 Context of socio-technical software analysis

The new emerging discipline of socio-technical software analysis should be viewed within the context of *meta-engineering* (meta-system engineering) and *meta-design*. Meta-system engineering covers such new systemic research and engineering fields such as meta-complexity, meta-knowledge, meta-ontology, meta-modeling as well as classical system theory and socio-cognitive engineering [20].

Meta-design [17, 18] is an emerging meta-system engineering methodology that 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 (1) increasing participation of users in system design process, and (2) 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 the socio-technical environments that empower users to engage actively in the continuous development of systems rather than being restricted to the 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.

Socio-technical software analysis is a part of meta-engineering methodology that focuses on the extraction of *meta-knowledge* and is a step preceding meta-design [9]. Meta-knowledge refers to knowledge that has been acquired and stored in prior system development activities and that is being applied to the current

software design project to improve the quality of the end product and to reduce its cost [29].

3.3 Aims of socio-technical software analysis

Socio-technical software analysis 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 aims of socio-technical analysis 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, analysis of external factors that affect software design.

The external factors (acc. to [28]) may include: (1) *Financial factors*, e.g. cost saving company may hire only recent graduates to develop software, consequently many mature programming practices are absent in developed source code. (2) *Policy factors*, e.g., large organizations often require that a particular OS or language be used on all projects. (3) *Communication factors*, e.g., the reduced cost of communication enables more extensive sharing of ideas between a large number of people involved on many levels and in numerous roles within a project. (4) *Cultural factors*, e.g. relationship between the younger engineers and the management impacts decision making and architectural freedom of the development team.

3.4 Comparison of socio-technical software analysis with traditional domain analysis

In general, *analysis* is the procedure by which we break down an intellectual or substantial system into parts or components. Domain analysis is a part of software engineering deals with analysis of complex, large scale software systems and the interactions within those systems, and results in the development of domain models (such as feature models or UML models). The results of domain analysis are used for developing required software system(s).

The socio-technical software analysis 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 the system designers and social (organizational) structure of the environment. Socio-technical analysis aims to understand complexity, interconnectedness and wholeness of components of systems in specific relationship to each other. In this aspect, it is similar to Systems Thinking [45].

Traditional analysis focuses on the separation and isolation of smaller constituent parts of the system (components) and analysis their interaction and relationship. In contract, socio-technical software analysis aims at expanding its view and including other related systems and domains in order to take into account larger number of interactions involved with the 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 domain analysis tends to involve linear cause and effect relationships. Socio-technical software analysis aims to include the whole complex of bidirectional interrelationships. 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 domain analysis methods.

Domain analysis methods often focus on revealing the quantitative characteristics of domain, such as metrics. Socio-technical software analysis focuses on revealing the qualitative characteristics of analyzed software systems, such as similarity, that are very much heuristic in nature and can be estimated only approximately. Thus, the outcome of the socio-technical software analysis is not unambiguous. It requires domain understanding and human decision.

Traditional analysis focuses on the behavior and functionality of designed domain systems (components, entities). The result is the data that characterizes domain systems (e.g., its features, aspects, characteristics, and metrics). Socio-technical software analysis continues the analysis 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 helps to reveal deeper properties of software systems that are usually buried in its source code or documentation.

Table 1: Comparison of views on a system of traditional and socio-technical software analysis

Aspect	Traditional analysis	Socio-technical software analysis
Type of system	Black-box (closed)	White-box (open, holistic)
View	Narrowing	Widening
Multiplicity	Stand-alone	Multiple
Boundaries	Clearly defined	Difficult to determine
Change	Static	Dynamic
Development	Use and dispose	Evolve and migrate
Object	Domain artifacts	Software
Target	Design guidelines	Socio-technical aspects
Characteristics	Quantitative	Qualitative
Focus	Behavior, functionality	Content

Socio-technical software analysis does not replace traditional domain analysis methods, but rather extends them for deeper analysis and domain knowledge. The comparison of traditional analysis and socio-technical software analysis is summarized in Table 1.

3.5 Socio-technical software analysis process

During his domain analysis and software development activities, the designer is influenced by the external factors such as its education, organization requirements and structure, technological environment, customer requirements and design constraints. Therefore such factors are reflected in the design artifacts. Design artifacts are products produced by software designers, such as domain models, source code, or documents, and analyzed during socio-technical analysis. During socio-technical software analysis, meta-knowledge is extracted from design artifacts. Meta-knowledge may have many forms such as meta-model, mental model, mind map, etc. It reflects social, organizational, psychological and technological structure and relationships of software design actors and processes.

The results of socio-technical software analysis (meta-knowledge) can be used for increasing quality of software products, improving software design processes, providing recommendations for better management of design organizations (team), raising the level of education, spreading good design practices and programming styles, improving workplace conditions, etc. The process of socio-technical software analysis is summarized in Figure 3.

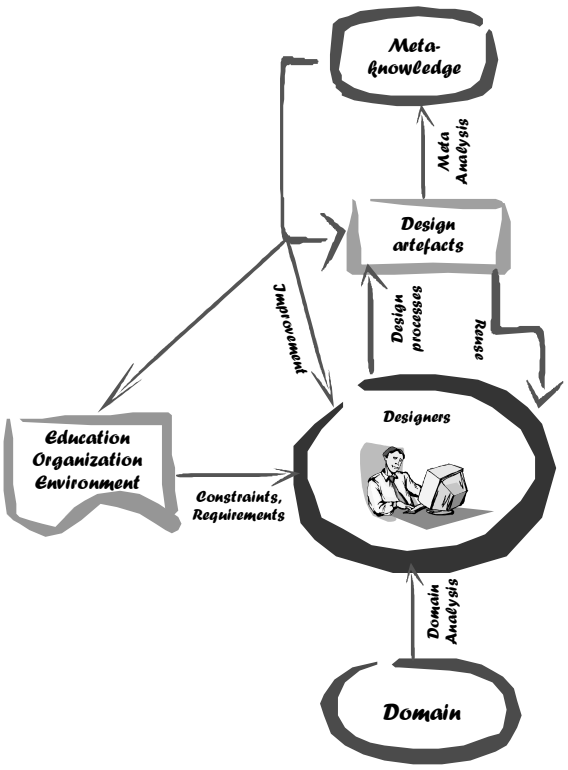

Figure 3: Socio-technical software analysis process

4 Case study

In this case study, we apply a typical socio-technical analysis method to estimate the level of code share (either by duplication or (re-)use) in programs created by a group of programmers, and to demonstrate how much social interaction and communication within a group contributed towards sharing source code in programmer programs.

4.1 Formulation of the problem

Our problem is to empirically establish, whether social relations have some influence on the program source code. When programmers work in groups, they inevitably communicate with each other, and share their ideas, programming practices and source code fragments. As a result their developed programs will be somewhat similar. If we take all programs developed by the group members as a single unit, there will be duplicated (redundant) code. Some of this redundancy is introduced by a programmer himself either by bad programming practices, code cloning using 'copy-and-paste' technique, or unintentionally using programming idioms related to the language or libraries; whereas, other similar code fragments can be attributed to code reuse or plagiarism. i.e. sharing of information. Our aim

is to estimate code duplication in programs and establish what part of it can be attributed to social information sharing within a group.

4.2 Description of case study

For our case study, we have selected computer science students attending "Introduction into information technologies and programming" course lectured at Kaunas University of Technology (KTU), Kaunas, Lithuania. The students were distributed into 4 groups, each having 8, 9, 9 and 5 members, respectively. The students were given the programming homework assignments in C++ language, which they had 4 weeks to complete. After completion of the assignment the program source code was collected and analyzed using a compression-based program redundancy metric based on the concept of Kolmogorov Complexity [33].

4.3 Description of used analysis method

Estimation of information redundancy in program source code is closely related to the concept of *information content*. There are several methods to evaluate information content such as computational complexity or Shannon entropy. Here we use the algorithmic metric of information content also known as *Kolmogorov Complexity* [33].

The main idea of Kolmogorov complexity is to measure the '*complexity*' (aka information content) of an object by the length of the smallest program that generates it. In general case, we have a domain object x and a description system (e.g., programming language) φ that maps from a description w (i.e., a program) to this object. Kolmogorov complexity $K_\varphi(x)$ of an object x in the description system φ, is the length of the shortest program in the description system φ capable of producing x on a universal computer such as a Turing machine:

$$K_\varphi(x) = \min_w \{\|w\| : \varphi_w = x\} \qquad (1)$$

Kolmogorov complexity $K_\varphi(x)$ is the minimal quantity of information required to generate x by an algorithm, and is the ultimate lower bound of information content. Unfortunately, it cannot be computed in the general case [33]. Consequently, one must approximate it. Usually, compression algorithms are used to give an upper bound to Kolmogorov complexity. Suppose that we have a compression algorithm C_i. Then, a shortest compression of w in the description system φ will give the upper bound to information content in x:

$$K_\varphi(x) \le \min_i \{C_i(\varphi_w)\} \qquad (2)$$

Now, as we have defined information content of the program, we must estimate the information redundancy of the program. For this, we can use the *Information Redundancy* metric R_φ [10] defined as follows:

$$R_\varphi(x) \le \frac{\varphi_w - \min_i \{C_i(\varphi_w)\}}{\varphi_w} \qquad (3)$$

Information redundancy R_φ represents the redundancy of information content (self-duplication) created by a separate member of the programmer's group.

Let $\Phi w(G)$ be the concatenation of all programs ${\varphi^j}_w$ developed by the members of group G. Let R_Φ be the information redundancy of Φ_w calculated acc. to Eq. 3.

Group information redundancy R_Φ represents the redundancy of shared information content created by all programmers within the group. It includes self-duplication as well as content shared with other members of a group.

We define the redundancy of shared information content created by two or more programmers within the group as: $R_\Phi(G) - \sum_j R_\varphi(\varphi_w^j)$, where ${\varphi^j}_w$ is a program created by programmer j in a group G using a description system φ.

We define the *Social information sharing metric* $S_\Phi(G)$ within the group G as follows:

$$S_\Phi(G) = 1 - \frac{\sum_j R_\varphi(\varphi_w^j)}{R_\Phi(G)} \qquad (4)$$

After applying Eq. 3 to Eq. 4, we obtain:

$$S_\Phi(G) = \frac{\sum_j \min_i \{C_i(\varphi_w^j)\} - \min_i \{C_i(\Phi_w^G)\}}{\Phi_w^G - \min_i \{C_i(\Phi_w^G)\}} \qquad (5)$$

$S_\Phi(G)$ is an estimation of source code related information sharing and exchange level in a social group as opposed to creativity and duplication of information by a stand-alone programmer. The larger is the value of $S_\Phi(G)$, the larger is the social cohesion of a group (i.e. the number of social connections within the group that result into the programming-related information exchange channels).

4.4 Results

For compression-based estimation of information content, here we use BWT (*Burrows-Wheeler Transform*) compression algorithm, because currently it allows achieving best compression results for text-based information [34] and thus allows to approximate information content and redundancy better. The results of the experiment with the Social information sharing metric (see Eq. 5) are summarized in Table 2.

The level of shared code was the smallest one in Group 4, because it had the least number of members, which means that there were fewer social interaction and communications, which had a direct impact on code sharing.

Table 2: Summary of experimental results

Group	No. of members	Avg. program size, B	Avg. compr. program size, B	Avg. information redundancy in programs	Information redundancy in group	Social information sharing
1	8	59.570	15.814	0.73	0.86	0.14
2	9	80.259	19.052	0.74	0.86	0.11
3	9	59.998	16.037	0.73	0.85	0.14
4	5	92.028	19.324	0.79	0.84	0.06

The total amount of original, shared and redundant (duplicated) information content in source code developed by the programmer groups is presented in Figure 4. It shows the amount of original information content created by the programmers, the amount of information content shared by the group members and the amount of redundant (duplicated) information. Original source code made about 17% of total code, while shared code made about 11%, and duplicated code made about 71% of total code.

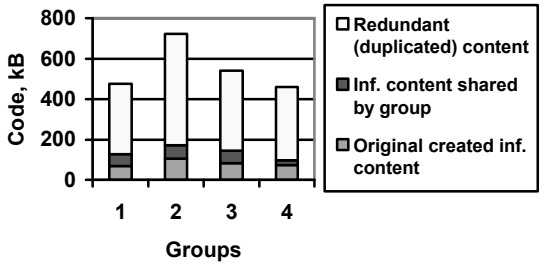

Figure 4: Duplicated, shared and original information content in developed programs

5 Discussion and conclusions

Software design processes and their artifacts have many perspectives: technological, social, psychological, etc. The socio-technical perspectives of software 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 and uncovered using the socio-technical software analysis methods.

The socio-technical software analysis methods attempt to uncover socio-technical information about software developers by looking at their produced output (source code, comments, documentation, reports) and by-products (tool logs, program traces, events, etc.). This paper has formulated four main steps of socio-technical software analysis (comparison for similarity, pattern discovery and extraction, generalization, interpretation) for extraction of valid and useful meta-knowledge from software design artifacts.

Socio-technical software analysis 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 the social aspects (how software was produced) from the purely technological aspects of software design, because they are mutually interdependent. The empirical methods used during socio-technical software analysis generate the results that are not always unambiguous and are open for further interpretations.

The application of the socio-technical software analysis methods provides 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 and knowledge sharing, the 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 the socio-technical software analysis 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, improving the performance of software design processes and quality of design artifacts (source code, documentation, etc.).

References

[1] Amrit, C., Hillegersberg, J., and Kumar, K. *A Social Network perspective of Conway's Law*, CSCW'04 Workshop on Social Networks, Chicago, IL, 2004.

[2] Bannon, L. *Developing Software as a Human, Social & Organizational Activity*, Invited Talk, 13th Annual Workshop on Psychology of Programming (PPIG'2001), Bournemouth University, UK, 2001.

[3] Booch, G. *Software Archeology*, a presentation given at the Rational Users Conference, 2004.

[4] Bryant, S. XP: *Taking the psychology of programming to the eXtreme*, 16th Annual Workshop on Psychology of Programming (PPIG'2004), Carlow, Ireland, 2004.

[5] Cannon-Bowers, J.E., Salas, E., and Converse, S. *Shared Mental Models in Expert Team Decision-Making*, in Castellan J. (ed.), Individual and Group Decision-Making: Current Issues. Lawrence Earlbaum and Associates, Inc., p. 221, 1993.

[6] Chong, J., Plummer, R., Leifer, L., Klemmer, S.R., Eris, O., and Toye, G. *Pair Programming: When and Why it Works*, Proc. of PPIG 2005, University of Sussex, Brighton, UK, 2005.

[7] Conway M.E. *How Do Committees Invent*, Datamation 14(4): 28-31, 1968.

[8] Curtis, B. *The Impact of Individual Differences in Programmers*, in Working With Computers: Theory Versus Outcome, Academic Press, 279-294, 1988.

[9] Damaševičius, R. *On the Application of Meta-Design Techniques in Hardware Design Domain*. International Journal of Computer Science (IJCS), 1(1): 67-77, 2006.

[10] Damaševičius, R. *Application of Meta-Analysis Techniques for Analyzing Socio-Technical Aspects of Program Source Code*. To be published.

[11] de Souza, C., Dourish, P., Redmiles, D., Quirk, S., and Trainer, E. *From Technical Dependencies to Social Dependencies*, Social Networks Workshop at the CSCW Conference, Chicago, IL, 2004.

[12] de Souza, C., Froehlich, J., and Dourish, P. *Seeking the source: software source code as a social and technical artifact*, Proc. of Int. ACM SIGGROUP Conf. on Supporting Group Work, GROUP 2005, Sanibel Island, FL, pp. 197-206, 2005.

[13] Douce, C. *Long Term Comprehension of Software Systems: A Methodology for Study*, 13th Workshop of the Psychology of Programming Interest Group (PPIG'2001), Bournemouth UK, 2001.

[14] Douce, C. *Metaphors we program by*, 16th Annual Workshop on Psychology of Programming (PPIG'2004), Carlow, Ireland, 2004.

[15] Espinosa, J., Slaughter, S., and Herbsleb, J. *Shared Mental Models, Familiarity and Coordination: A Multi-Method Study of Distributed Software Teams*, Proc. of 23rd Int. Conf. on Information Systems, Barcelona, Spain, pp. 425–433, 2002.

[16] Finholt, T. *Toward a social psychology of software engineering*, Perspectives Workshop: Empirical Theory and the Science of Software Engineering, Dagstuhl Seminar 04051, 2004.

[17] Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A.G., and Mehandjiev, N. *Meta-design: a manifesto for end-user development*, Communications of ACM 47(9): 33-37, 2004.

[18] Fischer, G., and Scharff, E. *Meta-Design—Design for Designers*, In D. Boyarski, W.A. Kellogg (Eds.): Proc. of the Conference on Designing Interactive Systems: Processes, Practices, Methods, Techniques, New York City, pp. 396-405, 2000.

[19] Foreman, J. *Product Line Based Software Development – Significant Results, Future Challenges,* Software Technology Conference, Salt Lake City, UT, 1996.

[20] Gadomski, A.M. *Toward the Identification of the Real-World Meta-Complexity*, Seminar "Interdisciplinarity in Research", Warsaw, 2004.

[21] Grinter, R.E. *Recomposition: Coordinating a Web of Software Dependencies*, Computer Supported Cooperative Work, 12(3): 297-327. Springer, 2003.

[22] Hales, D., and Douce, C. *Modelling Software Organisations*, In J. Kuljis, L. Baldwin & R. Scoble (Eds). Proc. of PPIG 2002, Brunel University, pp. 140-149, 2002.

[23] Hall, J.G., and Silva, A. *A requirements-based framework for the analysis of socio-technical system behaviour*. Proc. of 9th Int. Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ03), pp. 117-120, 2003.

[24] Herbsleb, J.D. *Metaphorical Representation in Collaborative Software Engineering*, Proc. of Joint Conf. on Work Activities, Coordination, and Collaboration, San Francisco, CA, pp. 117-125, 1999.

[25] Herbsleb, J.D., and Grinter, R.E. *Splitting the Organization and Integrating the Code: Conway's Law Revisited,* Proc. of Int. Conf. on Software Engineering. Los Angeles, CA, pp. 85-95, 1999.

[26] Hoc, J.M., and Nguyen-Xuan, A. *Language Semantics, Mental Models and Analogy.* In Hoc, J.M., Green, T.R.G., Samuray, R., Gilmore D.J. (Eds.), Psychology of Programming, pp. 139-156. London: Academic Press, 1990.

[27] Holt, R.C. *Software Architecture as a Shared Mental Model*, Proc. of the ASERC Workshop on Software Architecture, Paris, 2002.

[28] Hvatum, L., and Kelly, A. *What do I think about Conway's Law now?*, Conclusions of a EuroPLoP 2005 Focus Group, 2005.

[29] Kalfoglou, Y., Menzies, T., Althoff, K.D., and Motta, E. *Meta-Knowledge in Systems Design: Panacea... or Undelivered Promise*. Knowledge Engineering Review, 15(4), 381-404, 2000.

[30] Kerth, N.L. *On Creating a Disciplined and Ethical Practice of Software Archeology*, in OOPSLA 2001 Workshop Software Archeology: Understanding Large Systems, Tampa Bay, FL, 2001.

[31] Letovsky, S. *Cognitive Processes in Program Comprehension*, in Soloway, E., Iyengar, S. (ed.), Empirical Studies of Programmers, 58–79. Ablex Publishing Company, Norwood, New Jersey, 1986.

[32] Li, M., Chen, X., Li, X., Ma, B., and Vitány, P. *The Similarity Metric*, IEEE Transactions on Information Theory, 50(12), 3250-3564, 2004.

[33] Li, M., and Vitanyi, P. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag, 1997.

[34] Manzini, G. *The Burrows-Wheeler Transform: Theory and Practice*. Lecture Notes in Computer Science 1672, pp. 34-47. Springer Verlag, 1997.

[35] Marshall, L., and Webber, J. *The Misplaced Comma: Programmers' Tales and Traditions.* In J. Kuljis, L. Baldwin, R. Scoble (Eds.). Proc. of PPIG 2002, Brunel Univ., pp. 150-155, 2002.

[36] Merrill, M.D. *Knowledge objects and mental models*, Proc. of Int. Workshop on Advanced Learning Technologies (IWALT 2000), Palmerston North, New Zealand, pp. 244-246, 2000.

[37] Norman, D.A. *Cognitive Engineering.* In Norman, D.A., Draper, S.W., User Centred System Design. LEA Associates, New Jersey, 1986.

[38] Parnas D.L. *On the Criteria to be Used in Decomposing Systems into Modules*. Commun. of the ACM 15(12): 1053-1058, 1972.

[39] Robles, G., Gonzalez-Barahona, J.M., and Herraiz, I. *An Empirical Approach to Software Archaeology.* Proc. of 21st Int. Conf. on Software Maintenance (ICSM 2005), Budapest, Hungary, pp. 47-50, 2005.

[40] Rosen C.C.H. *The Influence of Intra-Team Relationships on the Systems Development Process: A Theoretical Framework of Intra-Group Dynamics.* In Romero, P., Good, J., Acosta Chaparro, E., Bryant, S. (Eds.), Proc. of 17th Annual Workshop on Psychology of Programming (PPIG 17), Brighton, UK, pp. 30 – 42, 2005.

[41] Sawyer, S., and Guinan, P.J. *Software Development: Processes and Performance.* IBM Systems Journal, 37(4): 553 – 569, 1998.

[42] Sillito, J., and Wynn, E. *Social Dependencies and Contrasts in Software Engineering Practice.* CSCW Workshop on the Social Side of Large-Scale Software Development, 2006.

[43] Trainer, E., Quirk, S., de Souza, C., and Redmiles, D.F. *Bridging the Gap between Technical and Social Dependencies with Ariadne.* Proc. of the 2005 OOPSLA Workshop on Eclipse Technology eXchange (eTX), San Diego, CA, pp. 26-30, 2005.

[44] Watt, S.N.K. *Syntonicity and the psychology of programming,* Proc. of 10th Annual Workshop for the Psychology of Programming Interest Group (PPIG 10), Open University, UK, pp. 75-86, 1998.

[45] Weinberg, G.M. *Quality Software Management: Systems Thinking.* Dorset House Publishing, 1991.

Weinberg, M.W. *The Psychology of Computer Programming.* Van Nostrand Reinhold, 1971.