Key Success Factors in Software Engineering
CEPIS, Council of European Professional Informatics Societies, is a non-profit organisation seeking to improve and promote high standards among informatics professionals in recognition of the impact that informatics has on employment, business and society.

CEPIS unites 36 professional informatics societies over 32 European countries, representing more than 200,000 ICT professionals.

CEPIS promotes

2 From The Editor’s Desk
Worth To Be Mentioned – Rafael Fernández Calvo

Monograph: Key Success Factors in Software Engineering (published jointly with Novática*)
Guest Editors: Luis Fernández-Sanz, Juan-José Cuadrado-Gallego, and Maya Daneva

3 Presentation
A Different Focus on Software Engineering – Luis Fernández-Sanz, Juan-José Cuadrado-Gallego, and Maya Daneva

5 Analysis of Software Engineering from An Engineering Perspective – Alain Abran and Kenza Meridji

25 Using UML™ 2.0 to Solve Systems Engineering Problems – Ian Barnard

39 Applying Service-Oriented Software Measurement to Derive Quality Indicators of Open Source Components – René Braungarten, Ayaz Farooq, Martin Kuzn, Andreas Schmietendorf, and Reiner R. Dumke

46 ISBSG Software Project Repository & ISO 9126: An Opportunity for Quality Benchmarking – Laila Cheikhi, Alain Abran, and Luigi Buglione

53 Requirements Elicitation in Web Engineering – Stephanos Mavromoustakos and Katerina Papanikolaou

59 The Human Factor in Software Engineering – Luis Fernández-Sanz and María-José García-García

Monograph of next issue (April 2006):

Visual Environments
(The full schedule of UPGRADE is available at our website)
Dear readers,

Since our previous issue (December 2005) the following newsworthy events have taken place.

"Informatik-Spektrum" joins UPENET (UPGRADE European NETwork)

"Informatik-Spektrum", founded in 1978, is a bimonthly journal published, in German, by Springer Verlag on behalf of the CEPIS societies GI (Gesellschaft für Informatik, Germany) and SI (Schweizer Informatiker Gesellschaft - Société Suisse des Informaticiens, Switzerland). It was created in 1978 and his website is at <springer.metapress.com/link.asp?id=101560>.

"Informatik-Spektrum", whose Editor in Chief is Hermann Engesser, becomes the sixth journal making part of UPENET.

Welcome!

New milestone of the UPGRADE Newslist

The UPGRADE Newslist has now more than 1,750 subscribers from all over the world, but mainly from Europe. The subscribers come from academia, business and public administration, offering so a vivid representation of the wide spectrum of readers of our digital journal.

The purpose of this list is to distribute relevant news about UPGRADE, such as publication of new issues, calls for papers, etc.

More information can be found at <http://www.upgrade-cepis.org/pages/editinfo.html>.

Italian edition of UPGRADE discontinued

In December 2001 an Italian edition of UPGRADE (summary, abstracts and some articles online) was born published jointly by the Italian CEPIS society ALSI (Associazione Nazionale Laureati in Scienze dell'Informazione ed Informatica) and the Italian IT portal Tecnoteca. Now this edition, available at <http://www.tecnoteca.it/upgrade>, has been discontinued.

We thank ALSI and Tecnoteca for their cooperation in this project and extend our most sincere gratitude to Roberto Carniel, who was the founder and editor of this version, for his valuable work. Roberto will stay with us as Associate Editor of UPGRADE.

Cordially yours,

Rafael Fernández Calvo
Chief Editor of UPGRADE and Novática
<rfcalvo@ati.es>
A Different Focus on Software Engineering

Luis Fernández-Sanz, Juan-José Cuadrado-Gallego, and Maya Daneva

Software Engineering (SE) has been driving the evolution of software development since the late sixties. So much has already been said about this discipline that we tend to think that yet another collection of contributions about topics so frequently addressed in journals or conferences is unlikely to offer readers a different perspective on the major challenges facing this discipline. Although every software developer has at least once wanted to see him or herself as somebody who performs development as a sound and genuine engineering process, a great many factors play a role in the success of the actual implementation of software engineering philosophy in daily practice.

We need to pay attention to a large number of issues if we want to experience the much vaunted benefits of an SE approach: software as a truly engineered product, the general improvement of software quality and user satisfaction, fewer delays and budget overruns, etc. Clearly software engineering is the way to achieve these objectives, but not all the concomitant problems stem from traditional issues involving technical matters or methodologies.

In this joint monograph of UPGRADE and Novática we have decided to focus on those topics which do not form part of the traditional core of typical SE books but are perhaps fundamental to the real success of SE both now and in the future.

This issue includes an interesting set of papers covering various key areas related to topics such as the application of SE principles to software development and maintenance in areas like education for an engineering discipline based on the SWEBOK (Software Engineering Body of Knowledge) framework; the efficient use of methodologies and notation (UML, Unified Modeling Language); quality assurance in specific areas such as open source components or quality data availability and conformance to standards of important project repository data such as ISBSG (International Software Benchmarking Standards Group); the human factor in SE projects from the viewpoint of standards and methods; the social and qualitative side of SE; and, last but not least, how to achieve an effective application of SE principles to web engineering. We have selected these topics because we believe them to be potential hot spots that are already influencing the future of SE.

"Analysis of Software Engineering from An Engineering Perspective", by Alain Abran and Kenza Meridji (École Polytechnique de Montréal, Canada, <alain.abran@polytechnique.edu>)

Key Success Factors in Software Engineering

The Guest Editors

Luis Fernández-Sanz received a degree in Informatics Engineering from the Universidad Politécnica de Madrid, Spain, in 1989 and a PhD in Computer Science from the Universidad del País Vasco, Spain, in 1997 (receiving a special mention for his doctoral thesis). Since 2000 he has been the head of the Computer Systems Dept. at the Universidad Europea de Madrid, Spain. Since 1992, he has been editor of the Software Engineering section of Novática, and co-editor and founder of REICIS (Spanish Journal of Software Quality, Engineering and Innovation, <http://www.ati.es/reicis/>) both published by the Spanish CEPSI society ATI (Asociación de Técnicos de Informática). He has guest-edited several monographs for Novática and UPGRADE, and has authored or co-authored several books about Software Engineering and software measurement, as well as a number of papers in international journals and conferences. He leads the Software Quality Special Interest Group of ATI in which capacity he has acted as chair of the Spanish Conference on Software Quality and Innovation organized by ATI. "luis.fernandez@uem.es"

Juan-José Cuadrado-Gallego is Full Professor at the Computer Science Dept. of the Universidad de Alcalá, Spain. Previously he was teaching in different universities in Spain and other European countries. He earned his PhD in Computer Science Engineering in 2001. His research activity is centered in the Software Engineering area, especially in software measurement and he is authoring of more than fifty scientific publications in journals, conferences and books. He is co-editor and founder of REICIS (Spanish Journal of Software Quality, Engineering and Innovation, <http://www.ati.es/reicis/>) published by the Spanish CEPSI society ATI (Asociación de Técnicos de Informática). Nowadays he belongs to the Information Engineering Research Unit at the Universidad de Alcalá, where he is the leader of the Software Engineering and Software Measurement research line, and also belongs to the Software Engineering Research Laboratory of the Université de Québec, Canada. "jjcg@uah.es"

Maya Daneva has been a Senior Research Scientist with the Information Systems Group, Department of Computer Science, University of Twente, The Netherlands, since November 2004. Prior to this, she collected 9 years of Enterprise Resource Planning implementation practice as an SAP process analyst at TELUS Mobility, the second largest wireless telecommunication company in Canada, where she was consulting on SAP reference architecture models, requirement engineering processes and requirements reuse metrics. Before 1996, Dr. Daneva was a Research Associate in the Institute for Information Systems, at the University of the Saarlandes, Saarbruecken, Germany, where she headed research projects with consulting companies on software process and product improvement and benchmarking. She has authored more than 50 papers published by IEEE Computer Society, ACM Press and Springer. She was nominated twice for best industry paper award at the International Conference on Requirements Engineering, 2003 and 2005. "m.daneva@utwente.nl"
problems support. Oriented metrics with an easy to use approach for tool provide a complete framework in which software quality in project deliverables. Quality models such as ISO 9126 Dumke Andreas Schmietendorf and the abovementioned ten by "seems to be based on emotional confidence. The paper check the quality of such code in daily practice, so its use unusual for there to be no structured or well-defined way to suspicion by industry as being defect-prone. In fact it is not open source components have usually been viewed with tors specially customized for Open Source components. Free of software measurement services to derive quality indica-
tors for all its phases. This approach obviously needs the sup-
port of tools, but it highlights the interesting relationship between models in UML 2.0 and the verification of arte-
facts and generation of test cases. In fact, it is actually guided through models (not a new idea) but the paper considers the problems for a truly practical application. The influence of the new version of UML on this process is also analysed. We included this contribution as a means of covering the role of UML and other methodological considerations in the present status of software engineering.

A group of researchers from the University of Magdeburg, Germany, under the leadership of Reiner R. Dumke, present a lightweight and on-demand composition of software measurement services to derive quality indicators specially customized for Open Source components. Free open source components have usually been viewed with suspicion by industry as being defect-prone. In fact it is not unusual for there to be no structured or well-defined way to check the quality of such code in daily practice, so its use seems to be based on emotional confidence. The paper "Applying Service-Oriented Software Measurement to De-

But quality control is not just limited to defect detection in project deliverables. Quality models such as ISO 9126 provide a complete framework in which software quality can be analysed. In may cases, software effort estimation tends to forget the major role played by quality as one of the three main factors for project management (time, effort, quality). The paper "ISBSG Software Project Repository & ISO 9126: An Opportunity for Quality Benchmarking", by Laila Cheikhi, Alain Abran and Luigi Buglione (École de Technologie Supérieure – Université du Québec, Montréal, Canada), provides an interesting analysis of the well-known ISBSG project data repository to check whether the principal quality characteristics covered by ISO 9126 are actually present in this data collection. This type of study focusing on the validity and usefulness of collected data is vital if we are to avoid something that is all too frequent in SE: the lack of reliable information on which to base decisions.

Web applications underwent an almost exponential growth in the late nineties in terms of volume, number, and importance, and now the term web engineering is firmly established as a reality for software practitioners. However, the special nature of web applications call for the customization and adaptation of analysis and elicitation processes. Stephanos Mavromoustakos and Katerina Papanikolaou (School of Computer Science and Engineering, Cyprus College) offer a new vision of needs analysis for web software projects as well as a number of important ideas to bear in mind when dealing with user re-

quirements in this type of projects. In fact the paper "Re-

quirements Elicitation in Web Engineering" is not so very far away from the social side of SE described by Dr. Dittrich in this issue.

Finally, "The Human Factor in Software Engineering", by Luis Fernández-Sanz and María-José García-García, (Universidad Europea de Madrid, Spain) is aimed at presenting an overview on how job profiles and positions for software development are covered by standards, methodologies and international job classifications. The paper presents the importance of this topic in the context of the influence that human resources have on software development productivity and quality. By way of a conclusion, the paper highlights the need to continue working on the clarification and unification of professional and educational frameworks, curriculum proposals, code of ethics and professional behaviour, job profiles, certification, etc.

We hope these contributions can be considered as a rep-

resentative selection of the emerging issues in the extremely broad field of software engineering. Obviously, despite their importance many areas have necessarily been omitted from this issue, but we believe that the quality of the authors and the interest of the content will be more than enough to sat-

ify the needs of our readers.

We will finish this presentation expressing our recogni-
tion to the authors by their valuable contribution, and to the editors of UPGRADE and Novática for the opportunity given of publishing this collective work.
Analysis of Software Engineering from An Engineering Perspective

Alain Abran and Kenza Meridji

Walter G Vincenti, in his book "What engineers know and how they know it", has proposed a taxonomy of engineering knowledge. Software Engineering, as a discipline, is certainly not yet as mature as other engineering disciplines, and some authors have even challenged the notion that Software Engineering is indeed engineering. To investigate this issue, Vincenti’s categories of engineering knowledge are used to analyze the SWEBOK (Software Engineering Body of Knowledge) Guide from an engineering perspective. This paper presents an overview of the Vincenti’s categories of engineering knowledge, followed by an analysis of the engineering design concept in Vincenti vs. the design concept in the SWEBOK Guide: this highlights in particular the fact that Vincenti’s engineering design concept is not limited to the design phase knowledge area in the SWEBOK Guide, but that it pervades many of the SWEBOK knowledge areas. Finally, the SWEBOK Software Quality knowledge area is selected as a case study, and analyzed using Vincenti’s classification of engineering knowledge.

Keywords: Engineering Knowledge, ISO 19759, Software Engineering, SWEBOK, Vincenti.

1 Introduction

"Engineering is a problem-solving activity...dealing mainly with practical problems" (Vincenti).

Software engineering is defined by the IEEE (Institute of Electrical & Electronics Engineers) as "the application of a systematic, disciplined, quantitative approach to the development, operation and maintenance of software, the application of engineering to software" (IEEE 610.12) [1]. Of course, in comparison with mechanical and electrical engineering, Software Engineering is still an emerging engineering discipline, and one that is not as mature as other classical engineering fields.

There are millions of software professionals worldwide, and software is a ubiquitous presence in our society. However, the recognition of Software Engineering as an engineering discipline is still being challenged.

Achieving consensus by the profession on a core body of knowledge is a key milestone in all disciplines, and has been identified by the IEEE Computer Society as crucial for the evolution of Software Engineering towards professional status. The Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), written under the auspices of the IEEE Computer Society’s Professional Practices Committee, was initiated in 1998 to develop an international consensus [2] in pursuing the following objectives:

- to characterize the content of the Software Engineering discipline,
- to promote a consistent view of Software Engineering worldwide,
- to provide access to the Software Engineering body of knowledge,
- to clarify the place, and set the boundary, of Software Engineering with respect to other disciplines, and
- to provide a foundation for curriculum development and individual certification material.

The SWEBOK Guide [2], also adopted as a technical report by the ISO (Internacional Organization for Standardization) [3], has been selected as the subject of a study to explore the following question: Is Software Engineering truly an engineering discipline?

The content of each knowledge area (KA) in the SWEBOK Guide was developed by domain experts and extensively reviewed by an international community of peers. This Delphi-type approach, while very extensive and paralleled by national reviews at the ISO level, did not specifically address the engineering perspective, nor did it provide a structured technique to ensure the completeness and full coverage of fundamental engineering topics. Therefore, it did not provide sufficient evidence that it had adequately tackled the identification and documentation of the knowledge expected to be present in an engineering discipline.

Authors

Alain Abran is a Professor and the Director of the Software Engineering Research Laboratory at the École de Technologie Supérieure (ETS) – Université du Québec, Montreal, Canada. He is the Co-executive editor of the Guide to the Software Engineering Body of Knowledge project. He is also actively involved in international Software Engineering standards and is Co-chair of the Common Software Measurement International Consortium (COSMIC). Dr. Abran has more than 20 years of industry experience in information systems development and Software Engineering. He is the co-executive editor of the IEEE project on the Guide to the Software Engineering Body of Knowledge – SWEBOK (ISO TR 19759). <aabran@ele.etsmtl.ca>

Kenza Meridji is a PhD student at the Software Engineering Research Laboratory of the École de Technologie Supérieure – Université du Québec, Montreal, Canada. She is pursuing research on the fundamental principles of Software Engineering. She holds a Master degree in Software engineering from Concordia University, Canada. <kenza.meridji.1@ens.etsmtl.ca>
In this paper, an approach is proposed to investigate the content of the SWEBOK Guide in a structured way to verify what engineering knowledge is included in the Guide, and what could be missing. This approach is based on Vincenti’s classification of engineering knowledge. However, since this classification of engineering knowledge had not, at the time of this investigation, been used to analyze other engineering disciplines, it was felt necessary to carry out some structuring and modelling of the embedded criteria to render its use practical in the analysis of the SWEBOK Guide. In particular, the engineering design concepts had to be probed further, since at first glance there seemed to be a disconnect between the SWEBOK Guide design concept and Vincenti’s description of engineering design. Finally, Vincenti’s criteria are used to analyze a section of the SWEBOK Guide, the Software Quality KA.

This paper is organized as follows: Section 2 introduces Vincenti’s engineering viewpoint, and Section 3 presents a set of models developed to facilitate the use of Vincenti’s concepts for the analysis of an engineering discipline. Section 4 presents a mapping of Vincenti’s engineering design concept to the SWEBOK Guide Software Engineering design concept. Section 5 analyzes, from an engineering viewpoint, the Software Quality KA of the SWEBOK Guide, and, finally, in Section 6, a summary and future research directions are presented.

2 Vincenti’s Engineering Viewpoint

2.1 Overview and Context

Vincenti, in his book "What engineers know and how they know it" [4], proposed a taxonomy of engineering knowledge based on the historical analysis of five case studies in aeronautical engineering covering a roughly fifty-year period. He identified different types of engineering knowledge and classified them in six categories:

1. fundamental design concepts,
2. criteria and specifications,
3. theoretical tools,

<table>
<thead>
<tr>
<th>Engineering Vocabulary</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>Denotes both the content of a set of plans (e.g. in the design for a new aeroplane and the process by which those plans are produced).</td>
</tr>
<tr>
<td>Normal configuration</td>
<td>The general shape and arrangement commonly agreed upon to best embody the operational principle.</td>
</tr>
<tr>
<td>Normal technology</td>
<td>According to Edward’s constant that “what technological communities usually do” comprises “the improvement of the accepted tradition or its application under new or more stringent conditions.”</td>
</tr>
<tr>
<td>Normal design</td>
<td>The design involved in normal technology. The engineer working with such a design knows at the outset how the device in question works and what its customary features are, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task. Normal design is evolutionary rather than revolutionary.</td>
</tr>
<tr>
<td>Operational principle</td>
<td>Defines the essential fundamental concept of a device. “How its characteristic parts... fulfill their special function in combination to [sic] an overall operation which archives the purpose.”</td>
</tr>
<tr>
<td>Production</td>
<td>Denotes the process by which these plans are translated into the concrete artefact.</td>
</tr>
<tr>
<td>Operation</td>
<td>Deals with the employment of the artefact in meeting the recognized need.</td>
</tr>
<tr>
<td>Radical design</td>
<td>How the device should be arranged, or even how it works, is largely unknown. The designer has never seen such a device before and cannot presume that it will succeed.</td>
</tr>
<tr>
<td>Engineering knowledge</td>
<td>The knowledge used by engineers. Engineering knowledge has to do not only with design, but also with production and operation.</td>
</tr>
<tr>
<td>Descriptive knowledge</td>
<td>The knowledge of how things are.</td>
</tr>
<tr>
<td>Prescriptive knowledge</td>
<td>The knowledge of how things should be to attain a desired end.</td>
</tr>
<tr>
<td>Device</td>
<td>Devices are single, relatively compact entities, such as aeroplanes, electric generators, turret lathes, and so forth (Laundan).</td>
</tr>
<tr>
<td>Systems</td>
<td>Systems are assemblies of devices brought together for a collective purpose. Examples are airlines, electric power systems and automobile factories.</td>
</tr>
<tr>
<td>Technologies</td>
<td>Denote systems and devices taken together.</td>
</tr>
<tr>
<td>Concepts</td>
<td>May exist explicitly only in the designer’s mind. They are unstated givens for the project, having been absorbed by osmosis, so to speak, by the engineer in the course of his development, perhaps even before entering formal engineering training. They had to be learned deliberately by the engineering community at some time, however, and form an essential part of design knowledge.</td>
</tr>
</tbody>
</table>

Table 1: Vincenti’s Vocabulary Relating to Engineering Terms and Concepts [4].
4. quantitative data,
5. practical considerations, and
6. design instrumentalities.

Furthermore, Vincenti stated that this classification is not specific to the aeronautical engineering domain, but can be transferred to other engineering domains. However, he did not provide documented evidence of this applicability and generalization to other engineering disciplines, and no author could be identified as having attempted to do so either. In this paper, we propose some pioneering work on the use of Vincenti’s categorization of engineering knowledge as constituting criteria for investigating Software Engineering from an engineering perspective.

The Vincenti categorization of knowledge was first used in a graduate seminar in 2002 at the École de Technologie Supérieure, Université du Québec (Canada), as an anal-

<table>
<thead>
<tr>
<th>Engineering Knowledge Category</th>
<th>Corresponding Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fundamental design concepts</strong></td>
<td>• About the design&lt;br&gt;• Designers must know the operational principle of the device&lt;br&gt;• How the device works&lt;br&gt;• Normal configuration&lt;br&gt;• Normal design&lt;br&gt;• Other features may be opened</td>
</tr>
<tr>
<td><strong>Criteria and specifications</strong></td>
<td>• Specific requirements of an operational principle&lt;br&gt;• General qualitative goals&lt;br&gt;• Specific quantitative goals laid out in concrete technical terms&lt;br&gt;• The design problem must be “well defined”&lt;br&gt;• Unknown or partially understood criteria&lt;br&gt;• Assignment of values to appropriate criteria&lt;br&gt;• This task takes place at the project definition level in the design hierarchy.&lt;br&gt;• Definition of technical specifications</td>
</tr>
<tr>
<td><strong>Theoretical tools</strong></td>
<td>• Mathematical methods and theories for making design calculations&lt;br&gt;• Intellectual concepts for thinking about the design&lt;br&gt;• Precise and codifiable&lt;br&gt;• They come mostly from deliberate research&lt;br&gt;• They are not sufficient by themselves</td>
</tr>
<tr>
<td><strong>Quantitative data</strong></td>
<td>• Specify manufacturing processes for production&lt;br&gt;• Display the detail for the device&lt;br&gt;• Data essential for design&lt;br&gt;• Obtained empirically&lt;br&gt;• Calculated theoretically&lt;br&gt;• Represented in tables or graphs&lt;br&gt;• Precise and codifiable&lt;br&gt;• They come mostly from deliberate research&lt;br&gt;• They are not sufficient by themselves</td>
</tr>
<tr>
<td><strong>Practical considerations</strong></td>
<td>• Theoretical tools and quantitative data are not sufficient. Designers also need practical considerations derived from experience&lt;br&gt;• Practical considerations are learned on the job, and not in school or from books&lt;br&gt;• Practical considerations are rarely documented&lt;br&gt;• Practical considerations are also derived from production and operation&lt;br&gt;• This knowledge is difficult to define&lt;br&gt;• This knowledge defies codification&lt;br&gt;• A prototype must often be built to check the designer’s work&lt;br&gt;• The practical consideration learned from operation is judgment&lt;br&gt;• Rules of thumb&lt;br&gt;• The practices from which these rules derive include not only design, but production and operation as well</td>
</tr>
<tr>
<td><strong>Design instrumentalities</strong></td>
<td>• Knowing how&lt;br&gt;• Procedural knowledge&lt;br&gt;• Ways of thinking&lt;br&gt;• Skills based on judgment&lt;br&gt;• Knowledge on how to carry out tasks&lt;br&gt;• Must be part of any anatomy of engineering knowledge</td>
</tr>
</tbody>
</table>

Table 2: Vincenti’s Engineering Knowledge Categories and Criteria.
Vincenital tool to tackle this issue by analyzing each of the SWEBOK KAs separately. The initial purpose of this approach was to gain insights into their content and structure, which, by definition, were expected to be of an engineering knowledge type. While it was easy for graduate students at the Master’s degree and doctoral levels to use Vincenti’s criteria to analyze the SWEBOK Design KA and to propose a mapping to the Vincenti categorization, this proved to be much more challenging for all the other KAs, to the point where some of these students questioned the relevance to be much more challenging for all the other KAs, to the point where some of these students questioned the relevance of the applicability of Vincenti’s categorization to these other SWEBOK KAs, and, as a corollary to that, that these other KAs did not necessarily constitute knowledge of an engineering type. The specific vocabulary defined by Vincenti was presented in Table 1.

2.2 Vincenti’s Categorization Criteria & Goals

Vincenital provides a categorization of engineering design knowledge and the activities that generate it. However, the divisions are not entirely exclusive; some items of knowledge can contain the knowledge of more than one category. From Vincenti’s definitions of each engineering knowledge-type category, a number of criteria were identified and have been listed in Table 2. The goals of each category have also been identified, and these are listed in Table 3.

3 Vincenti’s Classification of Engineering Knowledge Types

3.1 Relationship between The Various Categories

Since the categories are not mutually exclusive, it is important to understand the relationships between them. An initial modelling of Vincenti’s categories of engineering knowledge is presented in Figure 1. This figure illustrates that, in seeking a design solution, designers move up and down within categories, as well as back and forth from one category to another.

It can also be noted that three categories (theoretical tools, quantitative data and design instrumentalities) are related in the following manner: theoretical tools guide and structure the data, while quantitative data suggest and push the development of tools for their presentation and application – see Figure 2. Furthermore, both theoretical tools and quantitative data serve as input for design instrumentalities, while appropriate theoretical tools and quantitative data are needed for technical specifications. The next section presents several models to illustrate the relationships across these engineering concepts.

3.2 Vincenti’s Classification of Engineering Knowledge-type Models

We now present a detailed description of Vincenti’s six categories of engineering knowledge and the related models for each. Vincenti stated that these categorizations of engineering knowledge are not exclusive, since some elements of knowledge can be found in more than one category.

3.2.1 Fundamental Design Concepts

The goal of ‘fundamental design concepts’, according to Vincenti, is as follows: “Designers setting out on any normal design bring with them fundamental concepts about the device in question,” which means the definition of fundamental concepts related to the device by the designer. Fundamental design elements are composed of four elements (Figure 3): operational principles, normal configuration, normal technology and concepts in people’s minds. At first, these concepts exist only in the designer’s mind:

Concepts in people’s minds are inputs to the project (Figure 4).

<table>
<thead>
<tr>
<th>Engineering Knowledge Category</th>
<th>Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fundamental design concepts</td>
<td>Designers embarking on any normal design bring with them fundamental concepts about the device in question.</td>
</tr>
<tr>
<td>Criteria and specification</td>
<td>To design a device embodying a given operational principle and normal configuration, the designer must have, at some point, specific requirements in terms of hardware.</td>
</tr>
<tr>
<td>Theoretical tools</td>
<td>To carry out their design function, engineers use a wide range of theoretical tools. These include intellectual concepts as well as mathematical methods.</td>
</tr>
<tr>
<td>Quantitative data</td>
<td>Even with fundamental concepts and technical specifications at hand, mathematical tools are of little use without data for the physical properties or other quantities required in the formulas. Other kinds of data may also be needed to lay out details of the device or to specify manufacturing processes for production.</td>
</tr>
<tr>
<td>Practical considerations</td>
<td>To complement the theoretical tools and quantitative data, which are not sufficient. Designers also need less sharply defined considerations derived from experience.</td>
</tr>
<tr>
<td>Design instrumentalities</td>
<td>Besides the analytical tools, quantitative data and practical considerations required for their tasks, designers need to know how to carry out those tasks. How to employ procedures productively constitutes an essential part of design knowledge.</td>
</tr>
</tbody>
</table>

Table 3: Vincenti’s Engineering Knowledge Categories and Goals.
Operational principles define the essential fundamental concept of a device. "How its characteristic parts... fulfill their special functions in combination to [sic] an overall operation which archives the purpose." The operational principles must be known by the designers first (Figure 5) and constitute the basic components for the design, whereas operational principles are abstract, and the design moves from abstract concepts to precise concepts (Figure 6).

Normal configuration is "the general shape and arrangement that are commonly agreed to best embody the operational principle."

Normal technology is "the improvement of the accepted tradition or its application under new or more stringent conditions." Design, in Vincenti, "denotes both the content of a set of plans (as in the design for a new aeroplane) and the process by which those plans are produced." There are two types of design: normal design and radical design. The latter is a kind of design that is unknown to the designer, and where the designer is not familiar with the device itself. The designer does not know how the device should be arranged, or even how it works. The former is a traditional design, where the designer knows how the device works. The designer also knows the traditional features of the device. This type of design is also the design involved in normal technology, which was mentioned earlier. In conclusion, "normal design is evolutionary rather than revolutionary." Finally, a normal configuration and operational principles together provide a framework for normal design (Figure 7).

In Vincenti, a normal technology, or design, is part of a normal configuration and of a related operational principle.

3.2.2 Criteria and Specifications

The goal for ‘criteria and specifications’ can be expressed as follows: "To design a device embodying a given operational principle and normal configuration, the designer must have, at some point, specific requirements in terms of hardware." The designer designs a device meeting specific requirements which include a given operational principle as well as a normal configuration. First, the design problem must be well defined. Then, the designer translates general quantitative goals into specific quantitative goals (Figure 8): the designer assigns values or limits to the characteristics of the device which are crucial for engineering design. This allows the designer to provide the details and dimensions of the device that will be given to the builder. Furthermore, the output at the prob-

![Diagram](Figure 1: Vincenti’s Classification of Engineering Knowledge.)

**Operational principles** define the essential fundamental concept of a device. "How its characteristic parts... fulfill their special functions in combination to [sic] an overall operation which archives the purpose." The operational principles must be known by the designers first (Figure 5) and constitute the basic components for the design, whereas operational principles are abstract, and the design moves from abstract concepts to precise concepts (Figure 6).

**Normal configuration** is "the general shape and arrangement that are commonly agreed to best embody the operational principle."

**Normal technology** is "the improvement of the accepted tradition or its application under new or more stringent conditions." Design, in Vincenti, "denotes both the content of a set of plans (as in the design for a new aeroplane) and the process by which those plans are produced."

There are two types of design: normal design and radical design. The latter is a kind of design that is unknown to the designer, and where the designer is not familiar with the device itself. The designer does not know how the device should be arranged, or even how it works. The former is a traditional design, where the designer knows how the device works. The designer also knows the traditional features of the device. This type of design is also the design involved in normal technology, which was mentioned earlier. In conclusion, "normal design is evolutionary rather than revolutionary." Finally, a normal configuration and operational principles together provide a framework for normal design (Figure 7).

In Vincenti, a normal technology, or design, is part of a normal configuration and of a related operational principle.

![Diagram](Figure 2: Relationships between Theoretical Tools & Quantitative Data.)
lem definition level is used, in turn, as input to the remaining design activities that follow (Figure 9). These specifications are more important where safety is involved, as in the case of aeronautical devices. The criteria on which the specifications are based become part of the accumulating body of knowledge about how things are done in engineering. Finally, ‘criteria and specifications’ exists as a category of knowledge only in engineering and not in science. In science, the aim is to understand: scientists do not need to have highly specified or concrete objectives. In engineering, by contrast, to design a device, criteria and specified goals are crucial.

### 3.2.3 Theoretical Tools

Theoretical tools are used by engineers to carry out their design. The goal of the ‘theoretical tools’ category is expressed by Vincenti as follows: "To carry out their design function, engineers use a wide range of theoretical tools. These include intellectual concepts as well as mathematical methods" (Figure 10). Intellectual concepts (such as design concepts, mathematical methods and theories) are tools for making design calculations. Both design concepts and methods are part of science.

In the first class of theoretical tools are mathematical methods and theories composed of formulas, either simple or complex, which are useful for quantitative analysis and design. This scientific knowledge must be reformulated to make it applicable to engineering. The engineering activity requires that thoughts be conceived in people’s minds. In the second class of theoretical tools are intellectual concepts, which represent the language expressing those thoughts in people’s minds. They are employed first in the quantitative conceptualization and reasoning that engineers have to perform before they carry out the quantitative analysis and design calculations, and then again while they are carrying them out.

### 3.2.4. Quantitative data

The goal of ‘quantitative data’ is to lay down ‘the physical properties or other quantities required in the formulas. Other kinds of data may also be needed to lay out details of the device or to specify manufacturing processes for production.’ Besides fundamental concepts and technical specifications,
the designers also need quantitative data to lay out details of the device. These data can be obtained empirically, or in some cases they can be obtained theoretically. They can be represented in tables or graphs.

These data are divided into two types of knowledge: prescriptive and descriptive:

- Descriptive knowledge is "knowledge of how things are." It includes physical constants, properties of substances and physical processes. In some situations, it refers to operational conditions in the physical world. Descriptive data can also include measurement of performance.

- Prescriptive knowledge is "knowledge of how things should be to attain a desired end." An example might be: "In order to accomplish this or organize this, arrange things this way."

Operational principles, normal configuration and technical specifications are prescriptive knowledge, because they prescribe how a device should satisfy its objective (Figure 11).

3.2.5 Practical Considerations

According to Vincenti, the goal of ‘practical considerations’ is "to complement the role of theoretical tools and quantitative data which are not sufficient. Designers also need for their work less sharply defined considerations derived from experience." This kind of knowledge is prescriptive in the way that it shows the designers how to proceed with the design to achieve it. Vincenti refers to practical considerations as constituting non-codifiable knowledge derived from experience, unlike theoretical tools and quantitative data which are very precise and codifiable because these are derived from intentional research. This category of engineering knowledge is needed by designers as a complement to theoretical tools and quantitative data. These practical considerations are learned on the job, rather than at school or from books. They are not to be formalized or programmed. They are derived from design, as well as from production and operation. The practical consideration derived from production is not easy to define and cannot be codified, and a prototype is highly recommended to check the designer’s work (Figure 12). An example of a practical consideration from operation is the judgment that comes from the feedback resulting from use.

3.2.6 Design Instrumentalities

The goal of ‘design instrumentalities’ in the engineering design process required for the engineer’s tasks is "to know how to carry out those tasks. How to employ procedure productively constitutes an essential part of design knowledge".

Having the analytical tools, quantitative data and practical considerations at hand, designers also need procedural knowledge to
Key Success Factors in Software Engineering

4 The Design Process
4.1 The Engineering Design Process in Vincenti

According to Vincenti, the engineering ‘design’ concept "denotes both the content of a set of plans (as in the design for a new aeroplane) and the process by which those plans are produced". In Vincenti’s view, design is an iterative and complex process which consists of plans for the production of a single entity, such as an aeroplane (device), how these plans are produced, and, finally, the release of these plans for production.

Vincenti mentions that there are two types of design in engineering, **normal** and **radical**. In the former, the designer knows how the device works, how it should be arranged and what its features are. In the latter, the device is new to the engineer who is encountering it for the first time. There-

Figure 9: Problem Definition Level Output.

Design instrumentalities contain instrumentalities of the process, the procedures, judgment and ways of thinking. The latter are less tangible than procedures and more tangible than judgment; an example of ways of thinking is ‘thinking by analogy’. Judgment is needed to seek out design solutions and make design decisions (Figure 13).

Figure 10: Theoretical Tools Model.

carry out their tasks, as well as to know how to employ these procedures.

Theoretical tools contain intellectual concepts, mathematical methods and mathematical theories. Mathematical methods are used to provide language for quantitative analysis and design. Intellectual concepts employ physical reasoning, mathematical theories and mathematical theories and physical reasoning. Physical reasoning and mathematical theories and physical reasoning are employed in qualitative conceptualizing and reasoning.
fore, the engineer does not know how it works or how it should be organized.

He also mentions that design is a multilevel and hierarchical process. The designer starts by taking the problem as input.

The design hierarchies start from the project definition level, located at the upper level of the hierarchy where problems are abstracted and unstructured.

At the overall design level, the layout and the proportions of the device are set to meet the project definition. At level 3, the project is divided into its major components. At level 4, each component is subdivided. At level 5, the subcomponents from level 4 are further divided into specific problems.

At the lower levels, problems are well defined and structured. The design process is iterative, both up and down and horizontally throughout the hierarchy. Vincenti’s view of the levels of design is modeled in Figure 14. At each level of the hierarchy, a design can be either normal or radical.

4.2 The Engineering Process in The SWEBOK Guide

The SWEBOK Guide is composed of ten KAs -- see Figure 15. Each KA is represented by one chapter in the SWEBOK Guide.

4.3 Design Motion in The SWEBOK Guide

The Software Requirements KA is composed of four phases of software requirements: elicitation, analysis, specification and validation. The elicitation phase is the process of deriving requirements through observation of existing systems. Requirements specification is the activity of transforming the requirements gathered during the analysis activity into a precise set of requirements. Software Requirements Specifications describe the software system to be delivered. In the requirements validation phase, the requirements are checked for realism, consistency and completeness.

Software design is defined in [1] as both "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and "the result of [that] process". Software design in the Software Engineering life cycle is an activity in which software requirements are taken as input to the software design phase for analysis. "Software requirements express the needs and constraints placed on a software product that contribute to the solution of some real-world problem."

The result will be the description of the software architecture, its decomposition into different components and the description of the interfaces between those components. Also described will be the internal structure of each component.
4. It can be observed that it is defined significantly differently in the two documents; that is, design in engineering according to Vincenti is not limited to design as described in the SWEBOK Guide. In Vincenti, it goes far beyond the scope of the SWEBOK, that is: it is composed of the whole of the Software Engineering life cycle, as illustrated in Figure 16.

All the activities of the software life cycle, like the requirements phase, the design phase, the construction phase and the testing phase) map to a single phase in the engineering cycle, that is, design. These activities do not necessarily take place in the same order: for instance, testing in engineering starts right at the beginning, at the problem definition level, and goes on until the final release of the plans for production, while in the Software Engineering life cycle, as defined generically in the SWEBOK Guide, testing starts after the construction phase.

Figure 12: Practical Considerations Model.

Figure 13: Design Instrumentalities Model.

4.4 Design KA: Mapping between Vincenti and The SWEBOK Guide

The analysis of the term ‘design’ in both Vincenti and the SWEBOK Guide is presented in Table 4: it can be observed that it is defined significantly differently in the two documents.

Is there a direct and unique mapping of this ‘design’ term used in both the SWEBOK Guide and Vincenti’s categorization of engineering knowledge?

If there were such a direct mapping, would this mean that only the Design KA in the SWEBOK could be mapped to Vincenti’s engineering knowledge? Or, alternatively, is the notion of design defined by Vincenti different from the design concept in Software Engineering as defined in the SWEBOK Guide? And, if so, what is its scope within the SWEBOK Guide?

The definitions and descriptions of this term in both Vincenti and the SWEBOK Guide are presented in Table
The detailed mapping between the different design levels in engineering and in the Software Engineering life cycle is presented in Table 4.

5 Identification of The Engineering Concepts in The Software Quality KA

We present next an analysis of the engineering content within the SWEBOK Guide using one of its ten KAs as a case study, that is, Software Quality.

5.1 Software Quality

Authors and organizations have provided many different definitions of quality. For instance, Phil Crosby (Cro79) stated that it is "conformance to user requirements", while Watts Humphrey (Hum89) defined it as "achieving excellent levels of fitness for use". IBM uses the term "market-driven quality". Furthermore, ISO 900:2000 has described quality as "the degree to which a set of inherent characteristics fulfills requirements". Finally, the SWEBOK Guide introduces software quality as a separate KA, describing quality in different ways. The breakdown of software quality topics adopted in the SWEBOK Guide [2][3] is presented in Figure 17.

5.2 Analysis Using The Vincenti Classification of Engineering Knowledge

This section discusses the evaluation of the Software Quality KA of the SWEBOK Guide from an engineering perspective. To analyze the breakdown related to the Software Quality KA, the Vincenti classification of engineering knowledge is used to identify the strengths and weaknesses of this KA, and to gain further insights on the level of maturity of this topic from an engineering viewpoint.

This analysis is based on the models of engineering knowledge described earlier. These models give us a very
Design definition for engineering according to Vincenti

Design, as defined by Vincenti:

“The content of a set of plans (as in the design for a new aeroplane)” and “the process by which those plans are produced”.

Design definition for Software Engineering

Design is defined in [IEEE 610.12-90] as both:

“The process of defining the architecture, components, interfaces, and other characteristics of a system or component” and “the result of [that] process”.

Design, in the engineering life cycle is a process which starts by taking as input the problem, following a set of hierarchical levels. This process moves from problem definition to the production of a device as output.

Software design in the Software Engineering life cycle is an activity in which software requirements are taken as input to the software design phase for analysis. The result will be a precise description of the internal structure of the program.

<table>
<thead>
<tr>
<th>Requirement Specification</th>
<th>Design</th>
<th>Construction</th>
<th>Testing</th>
<th>Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4: Design According to Vincenti vs. Design in The Software Engineering Life Cycle.**

descriptive analysis of the various key elements contained in each of the corresponding engineering knowledge areas. This allows us to make an appropriate mapping between the different categories of the engineering knowledge area and software quality. It helps in identifying the engineering elements contained in this topic, as well as the missing ones. As a result, it looks into the software quality area from an engineering perspective. Table 6 describes the mapping between the corresponding criteria for the classification of engineering knowledge and the related software quality topics. This analysis can provide useful insights into possible strengths and weaknesses of the software quality topic. It helps categorize the knowledge contained in the Software Quality KA of the SWEBOK Guide: for instance, it covers all categories of engineering knowledge from an engineering viewpoint, but this does not mean that it is complete.

![Diagram](image-url)

**Figure 16: Design According to Vincenti vs. Design in The Software Engineering Life Cycle.**
Key Success Factors in Software Engineering

Table 5: Mapping of The Design Process in Engineering vs. The Software Engineering Life Cycle

<table>
<thead>
<tr>
<th>Levels</th>
<th>Description of the design process in engineering</th>
<th>Software engineering life cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Project Definition</td>
<td>Requirements</td>
</tr>
<tr>
<td>2</td>
<td>Overall design – component layout of the aeroplane to meet the project definition.</td>
<td>Specification</td>
</tr>
<tr>
<td>3</td>
<td>Major component design – division of project into wing design, fuselage design, landing gear design, electrical system design, etc.</td>
<td>Architecture of the system</td>
</tr>
<tr>
<td>4</td>
<td>Subdivision of areas of component design from level 3 according to the engineering discipline required (e.g. aerodynamic wing design, structural wing design, mechanical wing design)</td>
<td>Detailed design</td>
</tr>
<tr>
<td>5</td>
<td>Further division of the level 4 categories into highly specific problems</td>
<td>Construction</td>
</tr>
</tbody>
</table>

Figure 17: Breakdown of Topics for the Software Quality KA (SWEBOK Guide).

and inclusive.

6 Summary

The SWEBOK Guide documents an international consensus on ten Software Engineering KAs within what is referred to as an engineering discipline. Software engineering, as a discipline, is certainly not yet as mature as other engineering disciplines, and some authors have even challenged the notion that Software Engineering is indeed engineering. The work presented here has involved investigating this engineering perspective, first by analyzing the Vincenti classification of engineering knowledge, and second by comparing the design concept in Vincenti vs. the design concept in the SWEBOK Guide.

The result of this analysis was to show that the design issue in Vincenti is not limited to the design issue in the SWEBOK Guide: it goes beyond that, in that it is composed of the whole of the Software Engineering life cycle.

Finally, the SWEBOK Software Quality KA was selected as a case study and analyzed using the Vincenti classification as a tool to analyze this KA from an engineering perspective. This analysis was carried out to identify some of the strengths and weaknesses of the breakdown of topics for the Software Quality KA. It has shown that all the cat-
<table>
<thead>
<tr>
<th>Engineering Knowledge Category</th>
<th>Corresponding Criteria</th>
<th>Quality Concepts Refined</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fundamental design concepts</strong></td>
<td>• About the design&lt;br&gt;• Designers must know the operational principle of the device&lt;br&gt;• How the device works&lt;br&gt;• Normal configuration&lt;br&gt;• Normal design&lt;br&gt;• Other features may be created</td>
<td>• Planning the software quality process&lt;br&gt;• Quality characteristics of the software (QI), (QE), (QIU)&lt;br&gt;• Software quality models&lt;br&gt;• Quality assurance process&lt;br&gt;• Verification process&lt;br&gt;• Validation process&lt;br&gt;• Review process&lt;br&gt;• Audit process</td>
</tr>
<tr>
<td><strong>Criteria and specification</strong></td>
<td>• Specific requirement of an operational principle&lt;br&gt;• General qualitative goals&lt;br&gt;• Specific quantitative goals laid out in concrete technical terms&lt;br&gt;• The design problem must be “well defined”.&lt;br&gt;• Unknown or partially understood criteria&lt;br&gt;• Assignment of values to appropriate criteria&lt;br&gt;• This task takes place at the project definition level</td>
<td>• Quality objective to be specified&lt;br&gt;• Characteristics of quality tools&lt;br&gt;• Software characteristics&lt;br&gt;• Criteria for assessing the characteristics</td>
</tr>
<tr>
<td><strong>Theoretical Tools</strong></td>
<td>• Mathematical methods and theories for making design calculation&lt;br&gt;• Intellectual concepts for thinking about design&lt;br&gt;• Precise and codifiable</td>
<td>• Verification process model&lt;br&gt;• Formal methods&lt;br&gt;• Testing&lt;br&gt;• Theory measurement&lt;br&gt;• Verification/proving properties&lt;br&gt;• TQM (Total Quality Management)</td>
</tr>
<tr>
<td><strong>Quantitative data</strong></td>
<td>• Specify manufacturing process for production&lt;br&gt;• Display the detail for the device&lt;br&gt;• Data essential for design&lt;br&gt;• Obtained empirically&lt;br&gt;• Calculated theoretically&lt;br&gt;• Represented in tables or graphs&lt;br&gt;• Descriptive knowledge&lt;br&gt;• Prescriptive knowledge&lt;br&gt;• Precise and codifiable</td>
<td>• Quality measurement&lt;br&gt;• Experimental data&lt;br&gt;• Empirical study&lt;br&gt;• E.g. the process of requirement inspection&lt;br&gt;• Value and cost of quality</td>
</tr>
<tr>
<td><strong>Practical Considerations</strong></td>
<td>• Theoretical tools and quantitative data are not sufficient. Designers also need considerations derived from experience&lt;br&gt;• It is difficult to find them documented&lt;br&gt;• They are also derived from production and operation&lt;br&gt;• This knowledge is difficult to define&lt;br&gt;• It defies codification&lt;br&gt;• The practical consideration derived from operation is judgment&lt;br&gt;• Rules of thumb</td>
<td>• Application quality requirements&lt;br&gt;• Defect characterization</td>
</tr>
<tr>
<td><strong>Design仪nstrumentalities</strong></td>
<td>• Knowing how&lt;br&gt;• Procedural knowledge&lt;br&gt;• Ways of thinking&lt;br&gt;• Judgment skills</td>
<td>• Quality assurance procedures&lt;br&gt;• Quality verification procedures&lt;br&gt;• Quality validation procedures&lt;br&gt;• SQM process tasks &amp; techniques&lt;br&gt;• Management techniques&lt;br&gt;• Measurement techniques&lt;br&gt;• Project planning and tracking&lt;br&gt;• Quality assurance process&lt;br&gt;• Verification process&lt;br&gt;• Validation process&lt;br&gt;• Review process&lt;br&gt;• Audit process</td>
</tr>
</tbody>
</table>

Key Success Factors in Software Engineering

categories of engineering knowledge described by Vincenti are present in this KA of the SWEBOK; that is, it addresses the full coverage of all related engineering-type knowledge. This does not mean, however, that it is all-inclusive and complete, but only that the coverage extends to all categories of engineering knowledge from an engineering viewpoint.

The next stage of this R&D project will focus on investigating the application of Vincenti’s engineering knowledge to the analysis of a single candidate Software Engineering principle. This analysis will be performed from an engineering viewpoint to the primary processes contained in the SWEBOK Guide with respect to the following fundamental principle: "Manage quality throughout the life cycle as formally as possible".

References
Researching The Social Side of Software Engineering

Yvonne Dittrich

The social side of Software Engineering is still an under-researched area. Existing publications about qualitative social science methods and approaches are inadequate. This article gives an overview of the state-of-the-art in this developing field, identifies the central challenges and highlights ways to address them; how to make software development visible, how to combine the methods borrowed from Social Sciences with software process and method improvement, and how to handle the political side of this type of action research.

Keywords: Cooperative Work, Qualitative Software Engineering, Social Sciences, Software Process.

1 Introduction

"No rule is defining what it means to follow it." [39] And, in the same sense, no method defines what it means to apply it. So how do software engineers make sense of methods and apply them in concrete circumstances? How do they coordinate this deployment? How do they make process models work, how do they use planning and coordination tools? These questions address Software Engineering as a cooperative work practice, as a social endeavour.

The social side of software development has been identified from the very beginning as one of the key success factors [23][24][9]. Indeed, in the special track on Software Engineering for the next millennium, social science inspired research on the social aspects of Software Engineering was addressed as important future line of research [13]. Nevertheless, relatively little research is performed within the Software Engineering community and less is published in central journals or presented at the important conferences held so far. The social side of Software Engineering and qualitative research seems to take place mainly in workshops. [19][34]

The reason for this might be that the social science approaches which aim at understanding how people make methods, organisations and procedures work are mainly the qualitative ones. Also, qualitative methods are notoriously bad at producing isolated causal relationships that can easily be translated into software process improvement measures.

Qualitative methods allow the development of an understanding of the social side of software development from a members’ point of view. [29] By addressing Software Engineering as a social achievement, these methods provide rich descriptions of how software engineers make the software and its development work. Such research not only helps us to understand whether methods work or do not work, but also to understand how and why they work (see e.g. [11]). This understanding can then be used to improve methods and tools.

This article gives an overview of the existing research addressing the social aspects of Software Engineering published across different scientific discourses. Thereafter, the main problems of transferring social science methods into Software Engineering are summarised; the specific characteristic of software as an invisible, highly malleable, and complex product [5]; the fact that Software Engineering, as a discipline, aims at improving the development practice; and that therefore Software Engineering researchers are influencing their research subject in another way than the majority of the Social Sciences. Here we propose ways to address these difficulties. The article thus gives an overview of the state of the art and the difficulties of researching the social side of Software Engineering and points at ways to address these difficulties.

2 An Overview over Existing Research

As mentioned in the introduction, research on the social aspects of Software Engineering is distributed over different discourses. Roughly, one can distinguish the research published in the mainstream of Software Engineering, the research published as computer supported cooperative work, and a number of researchers often from Scandinavia publishing mainly in the information systems discourse. Of course these discourses partly overlap.

The intention of this article is not to give a complete overview of existing research but to give interested researchers and practitioners’ pointers to start further investigation of the research discourses presented here.

2.1 Qualitative Research in The Software Engineering discourse

One of the earliest and most widely cited qualitative studies on Software Engineering is ‘A field study of the software design process for large systems’ from 1988 where communication and coordination is indicated as the most crucial factors for software development [9]. The article cites research from the mid 70s and 80s both from the empirical...
research of programmers and the engineering management communities. Qualitative research has been part of the empirical research on Software Engineering from the very beginning. Nonetheless, it has been quantitative research that dominated and dominates the empirical research in Software Engineering.

Researchers addressing the empirical side of Software Engineering have so far mainly applied quantitative research methods.

Basilh, with his work on experimentation in Software Engineering and the Software Engineering Laboratory at the University of Maryland, laid the ground for this approach [1][2]. Software engineering research should develop models of and methods for software development and analyze and evaluate them.

One of a recent example of this kind of research is an article [18] on an empirical theory of coordination in Software Engineering.

However, not all aspects of communication and cooperation are easy to measure. Regarding this aspect, Seaman proposed to complement quantitative methods with qualitative ones. [32]

She regards qualitative research methods mainly as a complement in early research stages, a way to generate hypotheses that then can be tested quantitatively.

As the goal is to understand the mechanics of Software Engineering, i.e. the influence of the deployment of certain methods on the outcome of Software Engineering, the goal often is to identify quantifiable causal relationships. Seaman reports from qualitative observations of inspection meetings that are coded in a quantifiable way so that statistics can be applied.

Other research published in Software Engineering venues triangulates qualitative results with quantitative methods. One of the means for quality assurance of qualitative empirical research is the deployment of different methods and data sources or the cooperation of different researchers during the analysis to counter possible individual or methodological biases. [35] provides an example of this kind of triangulation, results from questionnaires, participatory observations, interviews and tool usage statistics were combined in order to understand work practices of software maintenance and compile a list of requirements for a software exploration tool.

Relatively few publications present purely qualitative research. [19] reports a study of coordination in distributed development based on interview data and document analysis [11]. reports how a traditional process model was used as a frame for iterative implementation and user-developer cooperation. [33] reports an ethnographic study of XP (Extreme Programming) practice as a culture that is based on and supported by the XP practices but reaches beyond ‘implementing’ them. All these studies have in common that they focus on understanding of how software engineers make the development work, rather than identify dependant and independent variables respectively relationships between them.

2.2 Software Engineering as Computer Supported Cooperative Work

As software development is a co-operative effort it has has become subject of discussion in Computer Supported Cooperative Work (CSCW) [31]. Traditionally empirical research in CSCW mainly uses qualitative methods: Ethnography, ethnomethodologically informed ethnography, for example, often combines with participatory design processes as action research.

Issues are; the use of representations and design work as embodied practice [37], organizational constraints and their influence on the work practice [6][7], the development of organizational patterns from within the project group [27][28][38] or the interaction of work practice and computer based tools [14][15]. The research focus here is on understanding the ways the members of a Software Engineering project achieve coordination of their cooperative effort. A more recent example of this kind of research is [30] where the authors explore how software engineers used plans to coordinate a widely distributed method and tool development project.

This kind of research is mainly rooted in ethnography. The researcher tries to understand software development as work practice from within. This sometimes leads to what, for software engineers, looks like the appreciation of a skillfully performed bad practice. The interesting question – what makes a disadvantageous practice less troublesome than changing the habit? – is seldom asked. Few authors seldom use the studies to further develop methods or tools for developers. Here the work of Grinter [14] provides an exception.

2.3 ‘Out of Scandinavia’

In reference to the seminal introduction of what non-Scandinavians call the Scandinavian Schools of Systems Development to the international research community [40] this section presents a regionally rooted strand of research, which is, internationally, mainly published in information systems venues. As the Scandinavian School on Systems Development is characterised by a humble attitude towards the expertise of the future users of the software under development, this strand of research takes the experience of the practitioners as a starting point [3][4][21][12].

Research is performed as Action Research meaning that researchers take part in industry in software development or software process improvement. The active participation is complemented by qualitative and quantitative data collection, and a detailed documentation of their own intervention.

This whole data provides the basis of the research focusing on the evaluation of the introduced measures. In [22] a major project on software process improvement is reported which involved researchers from different Danish universities and a number of industrial partners. This project resulted in a number of publications as well as addressing the social and organizational conditions for software processes and their improvements (e.g. [25]).
Due to the connection to the Scandinavian approach to systems development, the relation between user participation and Software Engineering methods and processes is a continuous thread of discussion within this community.

Already [3] discusses the adequacy of methods to support user developer communication. [28] addresses the constraints that software methods and processes put on developers regarding the possibility of usability into account. [16] reports on a case where agility in the development processes facilitates quick reaction to customer feedback, even in product development. [11] explores how the use of oriented software development can take place. In 2004 a special issue of the Scandinavian Journal of Information Systems addressed *inter alia* the implications of a change in the relationship between use and development of web-based systems. [8]

### 3 The Challenges of Qualitative Research

As discussed above, qualitative research, on the one hand, provides a better understanding the social of software development. On the other hand it faces special difficulties that have to be addressed when undertaking qualitative empirical research. Already quantitative research has to struggle with the problem that it is not easy to define simple measures of software as a product beyond lines of code or function points.

For observational methods the fact that software is invisible provides even more of a challenge. Doing research within an engineering discipline requires the focus to be not only on understanding what takes place, but also on generating improvements based on this understanding. This requires an extension of qualitative research into action research. And last but not least, doing qualitative fieldwork the researcher gets more involved with the social and political context in which Software Engineering takes place. This has to be reflected in both the design of the empirical work and the analysis of the results.

#### 3.1 Making Software Engineering Visible

Software is inherently invisible, and software engineers themselves work with different sets of representation, models, and specifications of which the source code of the final product is one [5]. This poses problems for all observational fieldwork methods. Observational methods rely on the coordination of activity that partly takes place through the joint artefact being observable in the manipulation of the artefact. This, in Software Engineering, is only partly possible, e.g. when observing design discussions in front of a whiteboard, as e.g. in [37].

However, this will only allow the observation of a small part of the communication mediated through the common artefact. Another possibility is to study the coordination of work through a common repository, a configuration management system e.g. But here the observation of the concrete action that leads to the changes in the files is difficult. Examples of this approach to making the complexity and cooperation of distributed work visible can be found in [18].

We ourselves experimented with techniques to map out the development process in cooperation with the involved practitioners [11]. Here the project members themselves took part in the task to represent the complex network of activities, documents and people involved in the development process. Here the map itself provides only part of the field material, the other extremely important part is the taping and transcription of the discussion which takes place when constructing the map with the involved practitioner. It also requires constructing a medium or notation for the representation that is geared towards the research question at hand. And it should be triangulated with other observational and interview methods, to counter possible biases by the involved practitioners, respectively to allow understanding the background for the specific way the members present their reality.

#### 3.2 From Understanding to Improvement

Qualitative social science methods are geared to make understandable the social aspects from a member’s point of view. This insight allows an understanding of how people (and in our case software engineers) manage to handle the cooperation and communication to achieve their task.

The effort involved in understanding the complexities of this achievement, e.g. the interface of project models, planning tools, social arrangements like role specific responsibility and meeting patterns for the coordination of a distributed development project [30], sometimes keeps the researcher from readily proposing tools or other means to help the problem’s practitioners in their daily practice. And it keeps researchers from stepping in the ‘bad practice’ trap that attracts many engineering researchers. "If the practitioners just would have used the tools and methods the right way, this would solve their problems" (see also [29]). The understanding of how methods, processes and tools actually inform the practice of software engineers is in itself an important contribution. However, as an engineering discipline, Software Engineering is not only interested in understanding how practitioners cooperate around the development of software, but also how this process can be supported with method, processes and tools.

Relating to the Scandinavian tradition of action research in participatory design we developed a research approach that designs and implements the improvement in an evolutionary cycle of empirical research, design of improvement measures together with the practitioners involved, and evaluation of the implemented improvements again with empirical means [10][12]. In this way we retain the strength of the qualitative empirical approach – understanding the social practice of software development from without – even through the improvement and method invention/adaptation process. Our experience so far supports the use of Cooperate Method Development frame for research cooperation with industry.

#### 3.3 Handling The Political Side of Research

Even experimental or quantitative Software Engineer-
ing research might be subject to social dynamics and politics within the field of enquiry as e.g. the failing trial to introduce elements of cleanroom software development in industrial practice reported in [2] shows. When doing qualitative research, the researchers gets involved with the practice they observe in a way that lets the ethical and political aspects of the research and the improvement proposals become more exposed.

Already the very act of seeing and representing how Software Engineering actually takes place can have a political dimension, e.g. when the research shows how developers circumvent mandatory procedures (see [36] for a more general discussion.)

The researchers’ positioning in relation to the political and power hierarchies within the organisation in which the research takes place will influence what the researchers get to see and what kind of improvements are possible (see [20] as an example for such influence and how it can be dealt with).

Being explicit about the researchers’ perspective, the ethical handling of the empirical data – e.g. rigorously implementing the member checking procedures agreed on – and being explicit about base and rational for the improvement proposals is therefore mandatory for empirical research in industrial practice.

4 Conclusions

Though qualitative research addressing the social side of Software Engineering is not yet visible as an identifiable strand of the Software Engineering discourse, we can identify a rich variety of research approaches and results. One major problem for this kind of research is that the publications are distributed over different discourses.

Few individual researchers are publishing in more than one discourse. References across the different discourses are infrequent. Researchers that publish in more than one community can serve as brokers between the different communities.

Qualitative research on Software Engineering is not easier than quantitative research. It provides a different set of challenges when implementing the research, both in terms of data collection, analysis, and argumentation. Qualitative research is not better than quantitative research. It addresses a different set of questions and provides complementing results. These complementing results can contribute to a richer understanding of how methods are used and why they work or don’t work. Taken together, the existing qualitative research on Software Engineering provides a sound base for mature research.

References
Key Success Factors in Software Engineering


[19] HSSE. Human and Social Factors of Software Engineering, held in conjunction with 27th International Conference on Software Engineering (ICSE 2005), 16 May, St. Louis, Missouri, USA, 2005.


Using UML™ 2.0 to Solve Systems Engineering Problems

Ian Barnard

Systems engineers are under ever-greater pressure to deliver increasingly complex systems. Systems engineers need better techniques for analyzing complex problems and for describing the complex systems that solve these problems. This paper discusses how UML 2.0 (Unified Modeling Language versión 2) can be used to address some of the issues faced by systems engineers. We explore techniques to specify system architectures, interfaces and behavior, using UML 2.0 examples.

Keywords: CASE Tools, Model-Driven Development, Testing, UML, Verification and Validation.

"Systems engineering is about creating effective solutions to problems, and managing the technical complexity of the resulting developments." [1]

1 Introduction

Systems engineers today are under ever-greater pressure to deliver larger and more complex systems, on time and within budget; to have any chance of success, the work has to be done right the first time.

This pressure will not diminish. Any way the activity can be refined through better techniques, and automated through better tools, must be grasped and maximized.

As problems and solutions become more complex, it is harder for authors to describe them, and harder for reviewers and other users to understand the descriptions. What is needed is a more powerful way of describing both problems and solutions, which can provide levels of abstraction, allowing designers to focus on design rather than worrying about implementation details.

As system specifications become more complex, it is harder to detect errors in the specification. What is needed here is a way to increase formality and precision, so that specifications are unambiguous and tools can automate checks for errors, removing the reliance on error-prone, expensive and all too scarce human beings to perform these tedious activities. This frees those people to focus on areas where their skills are really needed: the creative aspects of system development.

Every project uses pictures to help visualize aspects of development. How many times have you seen whiteboards similar to Figure 1, complete with that inevitable annotation: DO NOT ERASE? If pictures are so useful in helping understand and analyze problems, then perhaps taking them off the whiteboard and using them more formally as part of the development methodology is one way of improving the description of problems and solutions [2].

The Unified Modeling Language (UML) standardized by the Object Management Group (OMG), is a graphical language that has been very successful as a visual way of describing software. However, historically few systems engineers have used UML because it didn’t go far enough and it didn’t meet all their needs.

UML has evolved, though. The development of UML 2.0 was undertaken with the express intention of producing a language that has benefits for a much wider audience than just software developers - including the world of systems engineering.

Author

Ian Barnard has 20 years’ experience of systems and software development for systems ranging from embedded single-chip products for high-volume production to complex multi-processor systems for the Military market. During this time, he has gained extensive experience as Hardware Engineer, Software Engineer, and Team and Project Leader in many of the different areas that Telelogic’s products address and helped to introduce DOORS on a military systems engineering project. Since joining Telelogic in 1998, he has specialised in modelling, using SDL, UML 1.x and, since 2001, UML 2.0. Recent areas of involvement include combining UML modelling with requirements, and the DoDAF and MoDAF Architecture Frameworks. <barnard@telelogic.com>
This paper details the benefits that UML 2.0 can bring to systems engineering, by focusing on four questions which systems engineers would ask of UML 2.0.

### Status of the UML 2.0 standard

Development of UML 2.0 (Unified Modeling Language™) was formally initiated by the Object Management Group™ (OMG™) in 2000. The technical content of the standard was finalized in 2003 with the selection of the proposal from the U2 Partners consortium (of which Telelogic was a founder member and major contributor), and the finalized ‘superstructure’ standard which specifies the graphical diagram notation was released in 2004 [12].

#### 1.1 What about Systems Analysis and Structured Design?

For many years, systems engineers have used – with relative success – the techniques of Systems Analysis (SA) and Structured Design (SD). If you are familiar with notations such as Yourdon, Harel, Jackson and so on, then a lot of the concepts discussed in this paper will be familiar.

The successes of those approaches have been incorporated into the UML 2.0 notation [13]. The most significant objection to SA/SD has always been that the problem has to be approached from two different - and separate - viewpoints. SA starts with requirements and provides a way of describing them visually using Data Flow Diagrams (DFDs), and of decomposing them so that we fully understand the system that we intend to build. SD starts from the viewpoint that we need a good way to describe the final software solution in a way that we can understand and ultimately implement (i.e. using program structure charts).

Both techniques are valid. However, two different notations are used. Also, there is no automatic or even well defined way of translating DFDs and state charts into program structure charts. Experience helps us map one to the other, but there is still a gap that cannot be bridged by automation, so this is always the possibility of misunderstanding, and hence of introducing errors into the design: the process is inevitably manual, at a time when timescales are tightening and resources are scarce. As we will see, UML 2.0 helps in the areas of systems analysis and software design, but uses one common notation, so the issues of different notations and the resulting gap between the two disappear. More information of the possibilities for migrating from SA/SD to UML 2.0 is available elsewhere [13].

#### 1.2 Systems Engineering – What Are The Questions?

In virtually all aspects of systems engineering, formal documents use text. User requirements, interface descriptions and system design documents are all good examples. The written language allows great flexibility of expression, and can be quite precise if used carefully. However in many situations using pictures can make information significantly easier to capture and understand, supplementing the formal text. In systems engineering, gaining a quick and correct understanding of system requirements is fundamentally important, but of course it is very easy to draw informal pictures, so the first question considered here is: **Why choose UML 2.0 to visualize information?**

Increasingly, projects are split between teams and companies that are spread across countries and continents. This physical, logistical and contractual separation means that the participants need better ways to communicate all the detailed information involved, and to manage the impact of changes to that information. The second question considered here is: **How can UML 2.0 help create effective solutions?**

As systems become more complex, those involved need better facilities for describing complex problems and solutions, which can scale for use on both small and large projects. The issue is complexity, so the third question considered here is: **How can UML 2.0 help manage technical complexity?**

These aspects of UML 2.0 are valuable to systems engineers, but there may be other ways in which we might benefit from UML 2.0. The final question is: **How else can UML 2.0 help systems engineers?**

#### 2 Why Choose UML 2.0 To Visualize Information?

Pictures were used to convey information for tens of thousands of years before writing was invented. Creating pictures isn’t necessarily a simple process – that’s one way modeling tools help - but understanding them is easy because they are a basic, intuitive, form of communication.

There are many ways of representing information graphically. However, if each team, project or company were to choose its own graphical notation, the overall effect would be to make things worse. The difficulty of translating and communicating information between different notations is apparent. What is needed is a single, standardized, way of presenting information graphically, that is:

- Applicable for the top-down design approach needed to design complex systems.
- Precise in its syntax and semantics, so everyone knows unambiguously what the diagrams mean.
- Expressive and powerful, so it is applicable to a wide range of problems and hence projects.
- Abstract, so designers and architects can work at a higher level, freeing them to focus on the problems needing solving.

---

26 UPGRADE Vol. VII, No. 1, February 2006 © Novatica
Interchangeability will enable diagrams and complete models to be reused across different teams, projects and companies.

UML 1.0 didn’t satisfy these requirements at all well, which is one reason why systems engineers didn’t adopt it extensively. However, UML 2.0 has been designed with these characteristics in mind, and will satisfy the many modeling needs of systems engineers.

The OMG approach with UML 2.0 is defined by their Model Driven Architecture (MDA) framework[10], of which UML is a key component. The framework emphasizes the benefits arising from top-down design and automation. The OMG is working with INCOSE (International Council on Systems Engineering) to address how UML 2.0 can be used for systems engineering through the SysML standard [3][4].

3 How Can UML 2.0 Help Create Effective Solutions?

Two of the most basic problems faced in any project are communication and change management. UML 2.0 can help in both these areas.

Promoting Formal Communication

The ability to interchange formal specifications is of increasing importance. Today, projects are distributed between teams, between companies and between continents, which inevitably places stress on the quality of communication. Also, the growing complexity of systems requires ever more communication between all parties.

As more communication is required, the need for precise specification increases – particularly if there is a contractual relationship involved, because of the impact from errors due to misunderstandings. The UML 2.0 standard addresses this in two ways: the diagrams themselves have precise meaning, removing ambiguity; and, UML 2.0 will include a formal interchange specification (using XML, eXtensible Markup Language), so that interface specifications or complete models can be interchanged electronically, removing the possibility of errors in formal communication.

Promoting Informal Communication

Because of the visual nature of the UML 2.0 language, there is an immediate benefit for communicating ideas – it is very easy to understand. There are other visual notations, so one could argue that this benefit is not unique to UML. What is unique is the range of information that can be presented and the precision of the description in UML 2.0.

Managing Change

Change arises for all sorts of reasons: technology, competition, customer needs, errors, cost reduction, lack of time, etc. Change is inevitable, and rather than resist it, the only viable approach is for projects to embrace change and to accept that it will happen. The focus should be on managing change and minimizing the impact of making changes. A visual model in UML that is precise and easy to understand is also easier to change without introducing new errors.

Let’s look at some ways that UML 2.0 helps make information precise and easy to understand, by showing how it can be used for two aspects of systems engineering that are always involved when there is communication between teams and companies:

- Capturing and analyzing requirements, and
- Describing architecture and interfaces.

3.1 Capturing and Visualizing Requirements

The most basic part of trying to understand the requirements of a system is to first describe what it must do, by discovering the events to which it must respond, and what the response should be. This is the basis of use case or scenario analysis, which addresses functional requirements.

When starting to document functional requirements, there needs to be a way to capture the scenario details, but, as systems get more complex, the number of scenarios grows. It becomes essential to have a way to navigate through the scenarios. It also becomes more important that the information is easy to maintain and update. If some of the detail can be defined in one place and reused as needed, then the effort to maintain the specification is significantly reduced.

UML Use Case Analysis

UML use case analysis identifies the external entities (UML calls them actors, represented by stick figures) that interact with the system, and relates them using lines to the scenarios or use-cases they initiate or participate in (represented using ellipses), as shown in Figure 2; there can be many such diagrams in a complete model.

As important to note that actors are not necessarily human; they are often other systems, e.g. a network, a database, a radar sensor, etc.

The use case diagram is extremely simple, carrying little formal weight. It exists because it provides such an effective way of visualizing the
functional requirements on the system. It puts all the usage scenarios and actors into context and allows you to navigate to the detailed descriptions of the scenarios.

In Figure 2, the two actors *radar* and *missile system* interact with the *ProcessTrack* use case, which reuses (includes) another interaction, *CheckIFF*. This can also be included by other use cases, so that reusable definitions can be made in one place (for best maintainability) and referenced in many places (so the model is compact). The actual details of the use case, and how the reuse is made, are not specified in the use case diagram. This description is made elsewhere usually using another UML diagram, the sequence diagram.

It is essential that UML drawing tools also allow textual descriptions of use cases to be stored with the diagrams [5] because we are not dispensing with text: rather we are using a visual notation as a way of collecting, summarizing, representing and navigating through textual information.

### UML Detailed Behavioral Description

Representing detailed behavioral descriptions is the first area where UML has been significantly enhanced. The UML 2.0 sequence diagram notation can now describe complex behavior in a much more concise and powerful way, which is very effective for describing functional requirements.

The example in Figure 3 shows the detail of the interaction in the *ProcessTrack* use case. The entities interacting (called instances) are represented by rectangles containing the instance name with an attached vertical timeline. On a timeline, events are ordered by their sequence on the timeline, with time flowing down the page. Events are represented as horizontal lines with arrows between the timelines. Text attached to the events specifies the name of the event and any data carried by it, so it is possible to describe data used in a scenario.

All systems have states; for the scenario used throughout this paper, *Missile Fire Control*, the system might have

---

**Figure 3:** *ProcessTrack* Sequence Diagram.

**Figure 4:** Alternative Behavior Sequences.
states such as armed or passive. UML 2.0 sequence diagrams can include this state information when required. In the example in Figure 3, the implication of the two state symbols labeled armed is that both the RadarDataProcessor and the FireController must be in the armed state before the described sequence can happen. This provides essential information exactly where it is needed to understand the requirements. Figure 3 also shows how the reuse of behavior is made, by including a ref symbol containing the name of another interaction, in this case CheckIFF. Of course, several references may be made in one diagram, and a referenced sequence may itself contain further references.

**Alternative Sequences**

All systems have situations where at one time they respond to an event in one way, but respond differently at another time, maybe because of different data carried by the event or perhaps because of the state of the system.

UML 2.0 sequence diagrams include the ability to describe these alternative types of behavior within one diagram, as shown in Figure 4. The alt symbol can enclose any number of alternative sequences, each separated by a dotted line and else symbol. The alternatives can be complex, including references to other use cases or nested alternatives.

**And More …**

The sequence diagram can also show repeated, optional and exceptional sequences. Timers can be explicitly shown. Using alternatives, the variations of behavior needed to describe the handling of timers can be shown, such as the results of timeouts, and canceling of timers.

---

**Figure 5:** RadarDataProcessor Component Definition.

**Figure 6:** RadarDataProcessor Interface Definitions.
Summary – The Benefits of Use Case and Sequence Diagrams

Use case and sequence diagrams have many benefits as part of the specification of a system; they make the requirements easier to understand and easier to update.

However, the benefits don’t stop there. One of the ongoing activities at the OMG is to develop a testing profile for UML 2.0 that will allow sequence diagrams to be used for formal test specifications.

The benefits of using UML 2.0 to capture and analyze functional requirements are:
- Use case diagrams provide a high-level view of the system’s usage. Users can easily understand the range and context of scenarios, and they can easily navigate to detailed textual and graphical descriptions.
- The sequence diagram allows systems engineers to capture functional requirements visually, precisely and in detail, in a form that is intuitive, aiding formal communication and understanding.
- Reuse, through sequence diagram references, keeps detailed descriptions of system usage compact and easier to maintain.
- Sequence diagrams can be used as part of a formal process (e.g. - the definition of a test which can be executed). What was previously informal and informational can add formal value to the development activity, further helping to eliminate errors and misunderstandings.

3.2 Describing Architecture and Interfaces

Complete systems are invariably divided into smaller parts or subsystems during development. There are many motivations for this:

- To make the system easier to develop. It is usually easier to solve several smaller problems than one large problem.
- To benefit from re-use. Subsystems can often be reused from other projects, or reused in future projects, reducing the cost and time to needed to develop the complete system, and improving quality.
- To allow subdivision of the development project. Several smaller teams, which may be in different locations and/or companies, can work independently to develop the system.
- To reflect the physical construction or manufacture of the system.
- To provide logistical support. There may be downtime or reliability requirements, which may be satisfied only if the system is split into subsystems (sometimes known as configuration items) that can be replaced and/or maintained more easily and quickly than one monolithic system.

Using subsystems has many advantages, but systems engineers need tools to help maximize the benefits. For example, the strategy of splitting a large problem into several smaller ones can only succeed if independent teams can make rapid progress on these smaller problems. For this to be possible, there must be little ambiguity in the specification.

There is no room for errors or wasted time spent discussing the exact boundary between subsystems once the implementation activity starts.

There are two basic types of information required to proceed with development of a subsystem: what must it do and what are its interfaces?

If they sound familiar, it’s because these questions are the same as those asked at the system level when starting a project. UML use case and sequence diagrams can be used to help capture and analyze the requirements for subsystems, answering the question: what must it do?

However well these diagrams work to document examples of usage, they do not attempt to define interfaces completely. For example, they do not precisely define the information content of messages flowing in and out of the sys-

![Figure 7: Data Type Definitions.](image)

![Figure 8: Composite Structure (Architecture) Diagram.](image)
Key Success Factors in Software Engineering

So, how can UML 2.0 help create effective solutions?
UML 2.0 gives systems engineers a way to visually define the information they want to describe, such as architecture, interfaces, and functional requirements. However, this visual description is more than a collection of pictures:

- UML 2.0 diagrams have a precise and unambiguous meaning, making it easier to visualize requirements and specify architecture and interfaces.
- UML 2.0 diagrams are easy to understand, which aids development and maintenance.
- The diagram information builds a model that can be checked automatically.
- UML 2.0 models and diagrams can be interchanged electronically.
- Model-based development enabled by UML 2.0 reduces errors, and improves consistency, simplifying change and maintenance.

Defining Components (Subsystems)
All the components in a UML 2.0 model execute independently (asynchronously), communicating by passing messages through ports of the component. A component can only communicate with the outside world through its ports.

When a component is defined, its ports are also defined, as shown in Figure 5 for the definition of Radar Data Processor. Each port definition can have required and/or provided interfaces. A provided interface (denoted by a lollipop or ball symbol) accepts messages from outside the component flowing into it, whereas messages flow out of a required interface (denoted by a socket symbol). For example, in Figure 5, the RadarDataProcessor has three ports, denoted by small squares on the boundary of the class symbol. The radarPort port has a single provided interface Iradar, whereas the other two ports have both provided and required interfaces.

The definition of a subsystem is simple and clear; the port definitions describe where messages can flow in and out of the component, and the interface definitions on the ports define exactly what messages can pass.

Defining Interfaces
The definition of the interfaces (used in port definitions) can be made on the component definition, but it is usually more useful to define them elsewhere in the model using interface classes, as shown in Figure 6. This makes it easier to maintain the model, because the interface definitions are usually used in more than one place. By having a single definition that is reused by referencing it where needed, it is easy to make changes without introducing errors.

An interface definition contains a list of messages (UML 2.0 calls them signals), and defines their information content using a list of data types and names. For example, in Figure 6, the Iradar interface defines a single signal, radarMessage, carrying five parameters: IFFCode, Heading, Height, Speed, and Range.

The interface definitions are compact and precise; they very clearly specify the messages that together form the interface, and the information content of those messages.

Defining Data Types
The data types for message parameters are also completely defined in the UML model. For example, for the signal radarMessage defined in Figure 6, the types of the parameters are defined as shown in Figure 7. In this diagram, there are two enumerated types and two data structures.

This precise definition of the content of messages means that designers can unambiguously specify the information required by developers of subsystems and by implementers.

Assembling Subsystems into A System
How is architecture constructed? For any component, a UML Composite Structure diagram [11] (referred to here as an architecture diagram) such as Figure 8, assembles instances of other components, defining the internal structure of the component. On an architecture diagram, the ports of each instance are connected to ports on other instances, and to ports on the boundary of the component being produced. UML 2.0 tools will enforce checking to ensure that the provided/required interfaces of all instances are correctly connected. It simply shouldn’t be possible to create a definition where there are incorrectly matched interfaces.

In the example in Figure 8, three instances are interconnected through their ports, and are connected to the bound-
ary of the component. The resulting diagram is very simple to understand and to modify if the architecture changes.

The components assembled in architecture diagram may themselves be constructed from instances of other components. This is how you can describe a hierarchy that decomposes a large and complex system to smaller and more manageable subsystems.

But Why Bother with UML 2.0 To Describe Architecture?

Much of the basic information in the above diagrams could be described in words, using a conventional textual specification approach, maybe with some informal diagrams, so why is it worth using UML 2.0 architecture diagrams?

The standardized language and graphical presentation of the diagrams undoubtedly help communication and understanding. However, the benefits don’t stop there. To really use UML 2.0 effectively, it is necessary to use UML 2.0 design tools to create and manage the diagrams. Now, the information in the diagrams collectively describes a model of the system that we’re trying to describe. This brings significant additional benefits:

- Ease of design creation, reuse and maintenance,
- Ease of design navigation (e.g. - from a diagram to the definition), and
- Automatic error and consistency checking of the model.

These benefits are not available with text-based or informal approaches, and as systems get more complex, the value of these benefits increases. To help understand the benefits of true model-based development, imagine a maintenance manual for a car; the manual consists of text, pictures, drawings, tables, wiring schematics, etc. Now imagine that there are changes to the design of the car (i.e. changes to the UML model), and that the maintenance manual (i.e. the diagrams and documentation produced from the UML model) automatically updates to reflect the changes.

Now, maintaining the manual is much simpler and in fact it becomes difficult to introduce differences (i.e. errors) between the manual and the car. In the same way a truly model-based development means that changes to the model are automatically reflected in all the diagrams and documentation produced from the model, eliminating a significant source of errors when maintaining a system.

4 How Can UML 2.0 Help Manage Technical Complexity?

Increasing complexity of systems manifests itself in two ways: increasingly complex internal structure and increasingly complex behavior. As discussed previously, UML 2.0 architecture diagrams provide an effective way of describing complex architectures as a hierarchy of components (subsystems) communicating through defined interfaces.

However, there are two areas where increasing complexity raises concerns about how solutions are developed:

- How is the architectural hierarchy developed?
- How is complex behavior specified?

Underlying these issues is the ever-present concern: will it work? As complexity increases, so does the risk that errors or unforeseen deficiencies in the design will cause problems later in the project, during implementation and even during final testing. If UML 2.0 helps to improve the quality of system design to reduce the risk of failure, this is a major benefit to systems engineers.

4.1 Developing Architecture through Decomposition

As the design process proceeds, the initial architecture with just one or two layers of components is usually refined, by decomposing these components into further subcomponents, until sufficient detail has been added to allow implementation to start.

A logical approach to this process of developing a complete architecture is to take the functional requirements on the containing component, expressed in sequence diagrams, and from these develop refined sub requirements for the new subcomponents. Sequence diagrams could then show how the subcomponents interact to meet the requirements on the component.

UML 2.0 directly supports this approach. The sequence diagram notation includes the concept of decomposition.

This means that a UML 2.0 model using decomposition supports and enhances the development process by visualizing stages of development and allowing users to navigate through those stages within the model. By organizing the model in this way and reflecting how it is developed, UML 2.0 concretely addresses some of the technical complexity faced by systems engineers.

4.2 Specifying Behavior in UML 2.0

When developing a specification for a system, the objective is to satisfy the requirements, so designers need to specify how the system will realize the functional requirements described in use cases and sequence diagrams.

In UML 2.0 terms, this behavior is specified in the lowest level of the architecture hierarchy, in the components with no further decomposition. These components are what UML 2.0 calls active classes; they execute asynchronously from each other, and contain the logic that responds to input events, using a design approach called state machines.

UML 2.0 Behavior Specifications

The first stage of developing behavior specifications as state machines is to figure out what the states are, and how transitions are made between them. These form a very valuable part of a system specification because it is the information that distills all the sequence diagrams to satisfy the requirements, and it is the information that implementers use when developing the actual solution. UML 2.0 provides a state-centric state diagram notation, as shown in Figure 12 and Figure 13, which is ideal for use during the early stages of analysis, when the main problem is discovering the states. Later in the analysis, most of the states are known, and the problem becomes one of clearly specifying the detail of what happens during the transitions. The new transi-
Architecture and Sequence Diagram Decomposition

Initially, as shown in Figure 9, a sequence diagram is developed during analysis, documenting, say, the behavior of a system responding to events and interacting with the actors.

![Figure 9: Initial System-level Sequence Diagram.](image)

When the next phase of analysis starts, this diagram is updated, but only by amending the instance that is being decomposed. For example, in Figure 10, the instance MissileFireControlSystem, representing the system, is modified to add a reference to a new sequence diagram, ProcessTrack, which will describe the internal behavior of MissileFireControlSystem when it is decomposed.

![Figure 10: Updated Diagram References The Decomposed Sequence Diagram.](image)

Then a new sequence diagram is created to describe the internal behavior of the instance being decomposed, describing how it responds to the events received and then interacts with the other subsystems and actors. Figure 11 is the sequence diagram ProcessTrack, with three subsystems interacting to respond to the event radarMessage.

![Figure 11: Decomposed Sequence Diagram Details Internal Behaviour of System.](image)

4.3 But Will The Design Work?

As system behavior gets more complex, it gets more difficult to be confident that the envisaged solution will actually work once implemented. The only way you can be sure it’s going to work is by trying the design as early as possible in the lifecycle - preferably while you are creating...
Development of A UML 2.0 Behavior Specification

An initial (high-level) state diagram for the RadarDataProcessor subsystem might appear as in Figure 12. It starts at the state armed (as indicated by the arrow from the solid dot) and has one more state called passive. The response to a target detection event radarMessage is to send a message called engage. This is routed to the FireControl subsystem on the architecture diagram.

Figure 12: Initial State Diagram.

As the design develops, detail is added to the state machine. It might appear as in Figure 13, where the response to event radarMessage has been elaborated. It initiates a request (to the IFFDatabase) to identify friend or foe. If the response is isFoe, the engage message is sent (to FireControl). When the response engaged is received, the state machine returns to armed.

Figure 13: Partially Detailed State Diagram.

In the final stage, the detail of actions performed on transitions is added to the state machine, as shown in Figure 14. The transitions shown in Figure 14 are exactly the same as those in Figure 13, but with the detail of actions added. Now, the fully detailed behavior is specified completely visually, within the UML 2.0 model, and can be checked automatically for consistency and errors.

Figure 14: Part of The Fully Detailed State Machine.

Using traditional techniques, this is undertaken as additional development of software simulations, which is both costly and time-consuming. Frequently, you take calculated risks and simulate only certain, limited aspects of the design.

For the rest, you apply best-practice manual design and review, but then you have to cross your fingers and hope. When the system design changes, this often reduces the value of previous simulation work, or even invalidates it.

Updating the simulation to reflect changes and rerunning it is not easy because it requires another major development effort. Unless simulation redesign was planned into the project in advance, it is infeasible.

Hence, it’s common for system design errors to be discovered only during final testing of the just-assembled system. Or, even worse, while in use by the customer. Why? Because final testing or actual use is usually the first time the complete system can be fully exercised.
Often, the only way to make a system work at such a late stage is by making quick fixes; there isn’t time to go back through the design process and solve the problems properly. These fixes, implemented hurriedly to get the first system delivered without too much slippage, affect the system far into the future, often increasing the cost of in-service maintenance and enhancement for many years.

Studies of complex and software-intensive systems show that between 50 and 80 percent of the lifetime cost is incurred after delivery [6]. Decreasing this cost of maintenance is a significant opportunity – or a lifestyle for many businesses to reduce costs and to liberate scarce resources to generate more revenue: in other words to save money and to also make more money.

If there were a way, then, to actually check and test the design before it moves into implementation, errors could be eliminated, quality could be improved and maintenance costs reduced.

**UML 2.0 Checking**

The completeness of the UML 2.0 language brings with it the potential for greater automation of error checking even without executing the model.

Perhaps the most easily understood aspect of this is interface checking. Recall that every UML 2.0 model includes the specification of not just the operations which can be called on an object (i.e. the incoming interface) but also of the external operations it will call (i.e. the outgoing interface), as described above in Section 3.2.

Because this specification is available in UML 2.0 it is now possible to automate the checks that interfaces of interconnected objects are correctly matched in both directions, something which previously would only be found by time-consuming manual inspection, or – even worse – when the system was executing usually (hopefully) during testing but inevitably also in the field. Even this relatively simple check was not possible in UML 1.x, just as it is also not possible in languages such as C++ and Java which do not have the language constructs to specify both incoming and outgoing interfaces of a class; these can be constructed by enforcing some simple patterns, but this relies on the manual efforts and manually-enforced discipline of developers to achieve something which is automatable when using UML 2.0.

**UML 2.0 Simulation**

The ability to test a UML 2.0 model by executing it requires a way of compiling the graphical UML 2.0 model into an executable form; clearly this benefit can only be achieved by use of software tools, just as the ability to execute a program written in any textual (symbolic) language needs a software tool such as an assembler, or a compiler.

The key aspect here, though, is that model execution is possible with plain vanilla UML 2.0, whereas it wasn’t in UML 1.x. In the past many tool vendors have either extended UML 1.x to achieve model simulation or combined UML 1.x with other languages such as C++ and Java to add the missing behavioral detail needed to allow code generation to executable form. Some of these approaches continue today combining UML 2.0 with C++ and Java. However they all face problems, either with incompatibility between proprietary UML 1.x extensions for interchange, or with maintaining consistency between UML and (say) C++. In the latter case, error-checking is often either very limited or absent which means that many very basic error checks (e.g. a missing semicolon, or a reference in C++ to an attribute which has been renamed in the UML) are left to external compilers to detect after code-generation. And it isn’t uncommon for code generation (and hence C++ compiler checking of many types of error) to not be possible until very late in the project; this is little different from the original situation using textual languages alone, without UML, so these approaches don’t achieve anything like the advantages which can be achieved by using a single graphical language, UML 2.0, and compiling that language directly to executable form. But what can UML 2.0 execution offer? The most obvious benefit is that you can test the model using all the usual debugging techniques:

- Breakpoints which stop execution on a particular UML symbol within the detailed UML 2.0 behavioral specification (e.g. see Figure 14).
- Stepwise execution of the model to understand how alternative transitions execute
- Examination and modification of data structures such as attributes, message queues and timers.
- Capture sequence diagrams from the execution, for use as evidence of the testing undertaken, or to document the behavior of the model (see Figure 15).
- Capture execution coverage to understand the effectiveness of testing; answer the questions: how much of the detailed behavior has been executed? And how much more do I still have to test? For an example, see Figure 16..
- Regression testing, by capturing test sequences and replaying later, as frequently as needed.

The ability to verify that your visual model behaves as you expect is a big advantage of UML 2.0. And it doesn’t require programming in C++ or Java (whether inside or outside the model), or the development of test harnesses.

With UML 2.0, designers can specify their system in detail, and easily check and then test it during design creation. Errors and inconsistencies in the specification can be removed through automated checking, and automated testing can be used to eliminate dynamic errors.

As an example of the benefits of UML 2.0 when applied to software development, a major telecommunications manufacturer deployed Telelogic TAU®, a UML 2.0 model-driven development solution, to support the update of a network infrastructure system. The project involved challenges such as porting the system to new hardware and operating system platforms, a short development cycle (12 months) and a team of 100 developers distributed across multiple time zones. Telelogic TAU features include a model-based approach to design, executable models enabling early error detection, and parallel development support through its graphical compare / merge capabilities, and integration with configuration management.
tools. These capabilities helped the telecom manufacturer to reduce development costs and costs of inspections by a factor of 10, improve quality by eliminating 90 percent of coding errors and 30 to 50 percent of design errors, and reduce ramp-up time for new team member from six to nine months down to three.

Testing, Testing…1, 2, 3…

Properly specifying tests for a system is one of those oft-neglected areas of systems development; we all know we should do it yet there’s always something more important… And this is despite all the evidence that specifying tests is something which must be done early. Whether you are working in a venerable V model process, or applying agile development techniques, the imperative to specify and perform tests early in the lifecycle is clear.

UML 2.0 helps with this by enabling you to develop your test specifications within the same UML model as your system specification, using the OMG UML 2.0 Test Profile [8][9][14]. This means, for example, that testers will be able to use UML 2.0 to define the architecture of test systems and to define tests.

This is a big benefit to test developers. But if the project is also using UML 2.0 for systems engineering and/or for software development, there are even more advantages because of the ability to reuse common definitions across the project (eliminating many sources of errors) and the simplification of the complex process of ensuring that testing at all stages of systems development contributes to making top-level requirements-verification successful. Test specifications can be created using sequence diagrams, and are very easy to understand, to check, to test using model execution, to modify, and to reuse.

5 How else Can UML 2.0 Help?
The advantages of UML 2.0 for systems engineers, discussed previously, are very strong reasons for using the language, but the benefits needn’t stop there.

5.1 Components and Re-use
As discussed in Section 3.2., the architecture of a system is built from a hierarchy of components. These components are defined in one place, and referenced in the architecture hierarchy. But if a component can be defined in isolation and then assembled into one hierarchy (i.e. - the system you are developing), then it can also be assembled into another system in a different hierarchy, or into any number of systems. This is possible because the architecture diagram is not just a picture. The lines connecting ports precisely specify where signals are routed. They can, for example, drive automatic code-generation, so there is no need to manually modify the component when assembling it with other components; instead, the code generator automatically does the work. The result is true reuse of design, without unnecessary modifications common in "re-use" based on textual languages such as C and C++.

5.2 A Common Language with Software Developers
If your system requires software development, then UML 2.0 provides a common language. Systems engineers can use
Key Success Factors in Software Engineering

UML 2.0 to describe systems and software engineers can use UML 2.0 to develop detailed implementation, so UML 2.0 models can be shared or reused. This means that the gap between these two groups can be closed, and companies can benefit from the improved accuracy and reduction in errors as a result of automating the sharing of information.

5.3 A Modeling Language for Systems Engineers
The OMG has been working with the Systems Engineering community through INCOSE (International Council on Systems Engineering) to develop a profile (i.e. a customized version) of UML 2.0 specifically for Systems Engineers, called SysML. This adds, for example, the ability to graphically present the relationships between requirements and the model which satisfies them or justifies or explains their derivation.

6 Conclusions
We have seen the answers to four basic questions about UML 2.0:

- Sequence diagram decomposition mirrors the process of system development, by decomposing architecture
- UML 2.0 state machines allow complete visual specification of system behavior
- The use of a single language, UML 2.0, rather than (say) UML and C++ allows automatic checking for many potential errors in the specification, right from the start of the modeling activity
- The precision of UML 2.0 state diagrams allows them to be simulated without software coding, so the design can easily be tested, eliminating errors before implementation starts
- The UML Test Profile enables specification of tests and of test architectures using the same common definitions as the system model, and these can be checked and then tested against the system model long before the system itself exists

So, how does UML 2.0 help manage technical complexity?
UML 2.0 gives systems engineers a way to visually describe technically complex information:

- Sequence diagram decomposition mirrors the process of system development, by decomposing architecture
- UML 2.0 state machines allow complete visual specification of system behavior
- The use of a single language, UML 2.0, rather than (say) UML and C++ allows automatic checking for many potential errors in the specification, right from the start of the modeling activity
- The precision of UML 2.0 state diagrams allows them to be simulated without software coding, so the design can easily be tested, eliminating errors before implementation starts
- The UML Test Profile enables specification of tests and of test architectures using the same common definitions as the system model, and these can be checked and then tested against the system model long before the system itself exists
Why choose UML 2.0 to visualize information?

It provides a single, standardized, powerful language for precisely describing systems design and software design, which can be interchanged with other UML 2.0 users.

How can UML 2.0 help create effective solutions?

It gives systems engineers a way to visually define the information they want to describe, such as architecture, interfaces, and functional requirements. However, this visual description is more than pictures; it is based on a model which is precise and can be automatically checked.

How does UML 2.0 help manage technical complexity?

It gives systems engineers the means to visually describe technically complex information, in ways which reflect the development process, completely specify the behavior of complex systems and which can be automatically checked and tested during design creation, long before implementation starts.

How else can UML 2.0 help systems engineers?

It increases reuse within and between projects, through re-use of design components, and as a reusable common language between systems engineers (through SysML) and software developers and testers (through the UML 2.0 Testing Profile).

The UML 2.0 language allows systems engineers to benefit from a standardized visual notation that enables precise and powerful visualization of requirements, architectures and interfaces. These attributes help projects improve communication and minimize the impact of change by making models easier to understand and modify. As systems become more complex, UML 2.0 can help systems engineers by empowering them to describe these complex systems precisely and enabling them to routinely test the design as early as possible in the lifecycle - while creating it. UML 2.0 represents the next stage of evolution in the way that systems engineers work, helping them to get it right the first time, on time and on budget.

References

[12] "UML 2.0 is the official version of UML....": see <http://www.uml.org/#UML2.0>.
Applying Service-Oriented Software Measurement to Derive Quality Indicators of Open Source Components

René Braungarten, Ayaz Farooq, Martin Kunz, Andreas Schmietendorf, and Reiner R. Dumke

By addressing the industrial issue of streamlining by faster software development and, at the same time implying reduced cost and higher quality, component-based software development is currently very popular. Regrettably, in contrast with other component industries, software components which are supposed to increase quality are most often not certified while quality assurance is a very demanding task. This is especially the case with zero price components from the Open Source sector; these attract a great deal of industry’s attention but remain error prone. In this paper we present the idea of applying a light-weight and on-demand composition of software measurement services to derive quality indicators of especially Open Source components. Furthermore, our approach is then exemplarily applied using the OOMJ (Object-Oriented Measurement of Java Technologies) Web service to reveal quality indicators of Apache Tomcat 5.5.9 as Open Source component based on Chidamber & Kemerer (C & K) as well as Abreu’s MOOD (Metrics for Object Oriented Design) metrics set.

Keywords: Open Source Components, Quality, Service-Orientation, Software Measurement.

1 Introduction

In many conventional industrial sectors like automobile or consumer electronics, the diversified availability of domain-specific and most often standardized hardware components of certified quality is an essential prerequisite to long-term business success. Their production process can be better characterized as selecting and assembling components rather than developing a system from scratch. Unlike hardware components, the qualitative aspect of software components demands closer examination.

1.1 Software Components

The generally accepted definition of the term software component as per Szyperski [1] bears resemblance to the intuitive understanding of hardware components: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. [...] it can be deployed independently and is subject to compositions by third parties”. These components can be sold

Authors

René Braungarten is a PhD candidate and received a degree in Computer Science in 2004 from the University of Magdeburg, Germany. Since then he has been working as a research scholarship holder at the Software Engineering Group of the University of Magdeburg. His research interest includes software measurement databases and repositories. He is a student member of IEEE and IEEE Computer Society, an honorary member of DASMA e.V. and a member of the German Interest Group on software metrics. <braungar@ivs.cs.uni-magdeburg.de>

Ayaz Farooq is a PhD candidate and received a degree in Computer Science in 2005 from the University of Magdeburg, Germany. Since then he has been working at the Software Engineering Group of the University of Magdeburg. His research interest includes software measurement. He is a student member of IEEE and IEEE Computer Society. <farooq@ivs.cs.uni-magdeburg.de>

Martin Kunz is a PhD candidate and received a degree in Computer Science in 2004 from the University of Magdeburg, Germany. Since then he has been working as a research assistant at the Software Engineering Group of the University of Magdeburg. His research interest includes software measurement infrastructures. He is a student member of ACM and a member of the German Interest Group on software metrics. <makunz@ivs.cs.uni-magdeburg.de>

Andreas Schmietendorf received his PhD degree in Computer Science from the University of Magdeburg in 2001. He is currently working at the University of Applied Sciences in Berlin, Germany, as a professor with performance engineering as research field. Furthermore he is a consultant for T-Systems International in the area of IT-Service-Management. He works as a member for DASMA, the Central Europe Computer Measurement Group (CECMG) and the German Interest Group on software metrics. <andreas.schmietendorf@ba-berlin.de>

Reiner R. Dumke is currently working at the University of Magdeburg, Germany, as a professor with Software Engineering as research field. Dr. Dumke is one of the founders of the Software Measurement Laboratory (SML@b) of the Computer Science Department of the University of Magdeburg and co-editor of the Metrics News Journal. He is leader of the German Interest Group on software metrics and he works as a member of the COSMIC, DASMA, MAIN, IEEE and ACM communities. He received a degree in Mathematics in 1970 followed in 1980 by a PhD dissertation in Computer Science. He is the author and editor of more than 30 books about programming techniques, software metrics, metrics tools, software engineering foundations, component-based software development and Web engineering. <dumke@ivs.cs.uni-magdeburg.de>
as Commercial-Off-The-Shelf software products (COTS), or be developed in-house, or might be obtained free of charge from the Open Source area.

The manifold advantages of component-based development (CBD) mentioned by Andresen [2] include reusability of modules accompanied by their straightforward dimension that promotes rapid application development (RAD) as well as detachment of responsibilities achieved by an encapsulation of functional and technical aspects into different modules. Moreover, by permitting flexible software distribution and facilitating exchangeability due to specified interfaces, components bear the potential to be used in a company-wide and/or company-spanning context, thereby reducing cost aspects such as total cost of ownership (TCO).

These advantages have been recognized and taken up by the software industry to produce today’s component standards such as Sun Microsystems’ Java 2 Enterprise Edition (J2EE), Object Management Group’s CORBA Component Model (CCM), and Microsoft’s COM+ and .NET specifications.

1.2 Component Quality

In order to evaluate qualitative aspects of software components and/or a certain product in the sense of DIN EN ISO 8402 [3] circumscribing quality as "the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs", another ISO/IEC (International Organization for Standardization/International Electrotechnical Commission) standardization, namely ISO 9126 [4], assists. It puts quality aspects into six main categories, functionality, reliability, usability, efficiency, maintainability, and portability.

However, CBD is said to have the potential to increase the quality of software systems. Referring to this, Goulão and Abreu [5] pose that the reliability of components which have been used and tested in several cases, under a variety of conditions, is likely to be higher than that of software developed from scratch. But usually, only customers or final users can test the behaviour of components and assembled systems effectively - which can produce catastrophic results. It is a prolonged and complex task to assure the quality of in-house developed components, but it is an even harder task to do it with COTS, where source code or sufficient documentation are usually absent. Although Open Source components (OSC) bring along their source code, erroneous behavior might only appear when the entire system goes live. So IT organizations assembling systems using CBD and availing themselves of the Open Source area have an urgent need to identify quality indicators of those components and to identify potential flaws prior to a system’s delivery to the customer.

Because the basic approach of evaluating component quality does not seem to be viable for COTS due to the aforementioned reasons and can already be handled for in-house developed software, it turns out to be evident for to identify quality indicators for OSC. Thus, this paper turns its attention to the evaluation of OSC’s quality.

2 Peculiarities of Open Source Components

2.1 Open Source Characteristics

It is not sufficient to circumscribe the term Open Source by exploiting its virtual meaning of source code availability without charge as against the exclusive binary distributions of commercial vendors which ensures that all intellectual property is stashed away. While license agreements of commercial software are most often intended to limit customers in large areas requiring them to purchase additional licenses, the Open Source concept [6] aims at encouraging wide use by placing only a few generally accepted restrictions. Among these we will mention free redistribution as part of an aggregated software product, source code availability, allowance of modification under the same license, integrity of the author’s source code, ban on discrimination against any persons and/or groups or fields of endeavour, enforcement of license distribution alongside the source code as well as the prohibition of licenses specific to a product, to other software, or to technologies.

Open Source Creators

To ensure business continuity of organizations applying IT of any form their underlying software has to be future-proof and of high quality. This is not an imposition on volunteer Open Source developers. But as per Golden [7] that anxiety is in most cases without any reason, since volunteer developers strongly identify with their profession and consider programming as an important mode of self-expression. A survey of BCG (Boston Consulting Group) [8] tests that those developers, mostly males from Northern America or Europe, aged between 20 and 30, and with 11 years of software development experience, demonstrate an incredibly high interest in the contributed project’s continuity and success. While commitment of paid employees in commercial development projects is often overrated, volunteer Open Source developer’s passionate commitment can lead to great products.

Open Source Users

As a fact, Open Source is not only being used by small companies willing to stretch their IT budgets but by a number of major organizations such like Amazon.com or Google.com as constituent of their software infrastructure. [7] Withal, Open Source product’s user groups or communities can anonymously download products from Open Source portals like SourceForge or the developer’s Web site and take part in the development process by sending feedback.

2.2 Open Source Risks

Golden [7] describes four main problem areas that might deter companies from applying Open Source products: licensing, premature commitment, unchanging process, and the complexities of security and quality. But surprisingly, Golden suggests that organizations’ concerns about qualitative aspects of Open Source software are, for the most part, unfounded: "Far from our nightmare vision of poor quality code distributed by a flaky group of unqualified ide-
alists, we found that robust products were available that performed more than adequately”. One of the reasons could be that early and frequent releases are made publicly available to a large pool of users who quickly perform product examinations. Another reason is that source code availability facilitates higher quality of bug fixes by best of breed decisions and that direct user feedback ensures that the product implements only desired functionalities and does not suffer from commercial software’s problem of function creep to outrival competitors. To validate these general assumptions, it is sensible to quantitatively evaluate the quality of Open Source software components with the aid of software measurement.

3 Software Measurement Service-Orientation

At this stage, we want to briefly describe key concepts of service-orientation to reveal its advantages and substantiate our service-oriented approach for software measurement as cited as an example in section four.

3.1 Service-Oriented Architectures

Before discussing service-oriented architectures (SOA), the concepts of software architecture should be clarified by this rather classic definition of Bass, Clements, and Kazman [9]: “The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them”.

SOA Entities

Generally, service-oriented architectures can be characterized by the fact that they separate the implementation of the service from its interface. A “find, bind, and execute” paradigm enables a service’s customer to query a third-party registry for an adequate service implementation. If the registry finds a matching service, it provides the customer with a contract and an endpoint address. Following the notes of McGovern et al. [10], SOA configures its six entities, namely service consumers, providers, registries, contracts, proxies, and service leases after all, to support the above mentioned paradigm.

SOA Characteristics

According to McGovern et al. [10] SOA software can make use of any combination out of the following characteristics, (whereas e.g. Web services as one representative of SOA software do only implement a proportion of them): ability of being discoverable and dynamical binding, self-containment and modularity, interoperability, loose coupling, provision of network-addressable and coarse-grained interfaces, location transparency, composition on demand, and the ability of self-healing.

3.2 Service-Oriented Software Measurement

Measurement, in general, is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined unambiguous rules. Software measurement applies to a software engineering process thereby measuring numerous entities encountered along the way. According to Fenton and Pfleeger [11], software measurement is directed to three main components of software development: processes, products, and resources. Seen from another viewpoint, attributes belonging to these three aspects can either be internal types, that is directly and independently measurable, or external types, that is attributes only measurable from the outside... Following DeMarco’s [12] proverbial phrase “You can’t control what you can’t measure!”, as per Humphrey [13] among the basic reasons for collecting software measurement data are, obviously, support of understanding processes, evaluation of products or activities concerning specific acceptance criteria, controlling of processes, and calibration of estimation models to an organizational context. Being in most instances embedded in corporate measurement programs and/or quality assurance initiatives, multi-step measurement automation turns out to be a key factor [14] for cost effectiveness and developer’s commitment to the measurement process. To determine which measurement phases could actually benefit from automation and could thus be implemented by a service provider, we avail ourselves of Fenton and Pfleeger’s [11] high-level measurement context as illustrated in Figure 1 that alludes to four aspects of automation and/or service areas: Measurement itself [15], measurement collection [16][17], its collation and storage [16][17], and, finally, its analysis [18].

Since the functional scope and suitability to automation of a multiplicity of available commercial software metrics and estimation tools is limited [19] and Open Source approaches like Johnson’s Hackystat [20] or PROM of Sillitti
et al. [21] cannot respond to all-embracing measurement and analysis purposes, we propose a dynamic composition of software measurement services rather than hard-wired insular functionalities.

That procedure as illustrated in Figure 2 would enable a service consumer to flexibly tailor a measurement environment at runtime according to its instantaneous needs, in terms of the four service areas previously mentioned. The procedure would cover all aspects by leasing services with SOA characteristics from a service provider via a registry as long as needed and combining them with the aid of a contract, in contrast to investing in inappropriate and expensive conventional, hard-wired measurement software suites.

4 Case Study Using the OOMJ Web Service

This section presents a case study for a software measurement service being in line with the proposed SOA for software measurement. Withal, the introduced "Object-Oriented Measurements of Java Technologies" (OOMJ) Web service, along with a simultaneous software measurement Web application, is envisaged to provide an approach to evaluate Java components and reflects an adequate case study to evaluate OSCs. Even though this case study confines to Java-based OSCs, it provides meaningful results because of the sweeping pervasiveness of Java in the Open Source area.

4.1 SOA-Related Architecture Aspects

The OOMJ Web service and Web application attempts to cover the aforementioned service areas by providing a measurement view of Java technologies based on Chidamber & Kemerer (hereinafter called C&K) [22] and MOOD (Metrics for Object Oriented Design) [23] metrics sets, enabling users to quantitatively analyze Java technology libraries, measure and analyze online any Java library and/or OSCs, and provide measurement results in a portable form for further customized view and analysis. Although the proposed service-oriented architecture for software measurement implies one service for every single service area, OOMJ does provide its functionality altogether providing one interface to the system of components as illustrated in Figure 3. A future trend, and the next advancement of OOMJ, will be segmenting the combined services approach towards a single service implementation for every service area.

4.2 Measurement Exploration

Extract of Measurable OSCs

In addition to measuring any in-house developed component, the OOMJ Web Service provides measurement and analysis for a vast set of existing Java libraries. These include Open Source libraries provided by Apache Tomcat Servlet Container (Tomcat 5.5.9, 4.1.31), Apache projects (BCEL 5.1, Cactus 1.7, Lucene 1.4.3), Java-XML libraries (Dom4j 1.6.1, 1.5.2), chart and graph libraries provided...
by JFreeChart 1.0.0 and Cewolf 0.10.3, a unit testing library called HtmlUnit 1.6, as well as several other standard Java libraries provided by J2SE 1.5.0, J2EE (versions 1.4, 1.3.1, 1.2.1), J2ME, JWSDP 1.5 and Java Card 2.2.1.

Measurement Using OOMJ

Figures 4 and 5 show sample screen shots from the OOMJ Web application. It is unique in being freely available online and providing extensive metrics results in portable XML format for further customized analysis or otherwise use.

Exploring Quality Indicators

C & K and MOOD metrics sets highlight peculiar features of a software design. C&K metrics suite consists of six class-level metrics, namely depth of inheritance tree (DIT), number of children (NOC), weighted method per class (WMC), coupling between objects (CBO), response for a class (RFC) and lack of cohesion between methods (LCOM). MOOD metrics set consists of method/attribute hiding factor (MHF/AHF), method/attribute inheritance factor (MIF/AIF), polymorphism factor (POF) and coupling factor (COF).

Several empirical validations of these metrics sets exist in literature which have identified correlation between these metrics and several quality indicators like maintainability, fault-proneness, change-proneness, productivity and complexity. Results of these metrics evaluations for a software component can be analyzed using a variety of statistical techniques to indicate possible quality aspects of the underlying software design. We restrict our attention to measurement results for Apache Tomcat 5.5.9 library. Table 1 provides results of 95% confidence intervals for C&K metrics for this library. A wider variety of results is available in [24].

The results indicate that depth of inheritance tree (DIT) is quite small for most of the classes. Inheritance depth has been found to be correlated to maintainability [25][26]. Deeper classes tend to increase complexity and are thus avoided. Number of children (NOC) also shows similar distribution as compared to DIT, rather a higher percentage of classes tend to have a zero NOC value. A higher number of children means a higher reuse but, on the other hand, it also shows an improper class abstraction. Classes having higher NOC value are more critical to the design and need rigorous testing. Weighted method per class (WMC) affects class complexity and this effect is transferred to inheriting classes.

Consequently, more effort and time are needed for maintenance and testing. WMC is also found to be correlated to fault-proneness [27] and is close to 12 for this library. Extremely high values of WMC should trigger designer’s attention for possible rework.

Coupling between objects (CBO) affects change proneness [28] and higher inter-object class coupling calls for rigorous testing [29]. Therefore, a smaller CBO is advocated. It is observed to be close to 8. Response for a class (RFC) is similar kind of coupling measure which focuses on coupling between classes. A higher class coupling reduces its testability and understandability. The RFC for the mentioned library ranges between 22 and 27. LCOM is a measure of lack of cohesion between methods inside a class. Chidamber et al. [30] found that high LCOM value was associated with lower productivity, greater rework and greater design effort. The probable value of LCOM for the observed library is between 54 and 85.
Table 2 gives value of 95% confidence interval calculated for MOOD metrics for Apache Tomcat 5.5.9 library files.

Method hiding factor (MHF) and attribute hiding factor (AHF), part of MOOD metrics set, measure the level of encapsulation implemented by a system. Encapsulation copes with complexity and helps develop maintainable and reusable software. Tomcat 5.5.9 shows a very high level of encapsulation with both MHF & AHF higher than 98%. Ideally, all attributes should be hidden and AHF in this case is close to the optimal value. MHF is also quite high for these set of libraries which seems anomalous. A very high value indicates little functionality while a low value indicates improper encapsulation. This very high value of MHF has been observed for other Java libraries as well. A possible explanation could be that the value is numerically true and is correctly calculating ratio between visible and total available methods in the system but is not actually measuring level of encapsulation.

This is supported by a critical analysis of MOOD metrics set given in [31]. Inheritance is measured by method inheritance factor (MIF) and attribute inheritance factor (AIF). Inheritance on one hand results in reuse but on the other hand may add to complexity of design. MIF may range between 3 to 30% while AIF shows probability of value between 6 and 45% for the observed library. Polymorphism factor (POF) is a measure of the level of polymorphism implemented by a system. It allows simplicity by providing dynamic binding but may also complicate tracing control flow within classes. A low value of around 0 to 6% is observed in most of the cases. Again, detailed results can be found in [24].

5 Conclusions
In this paper we have presented the idea of applying a lightweight and on-demand composition of software measurement services to derive quality indicators of, especially, OSCs. Our approach has then been applied exemplarily using the OOMJ Web service to reveal quality indicators of Apache Tomcat 5.5.9 as OSCs based on C&amp;K as well as Abreu’s MOOD metrics sets to evaluate its quality. In doing so, our proposed service-oriented measurement approach turned out to be an interesting and further-to-be-explored alternative to conventional software measurement suites containing merely a hard-wired set of functionalities.

Table 2: 95% Confidence Interval: Mean of MOOD Metrics for Apache Tomcat 5.5.9.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Lower Limit</th>
<th>Upper Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHF</td>
<td>99.14</td>
<td>99.98</td>
</tr>
<tr>
<td>AHF</td>
<td>97.68</td>
<td>99.97</td>
</tr>
<tr>
<td>MIF</td>
<td>3.76</td>
<td>29.64</td>
</tr>
<tr>
<td>AIF</td>
<td>6.29</td>
<td>44.45</td>
</tr>
<tr>
<td>POF</td>
<td>0.12</td>
<td>5.71</td>
</tr>
</tbody>
</table>

Table 1: 95% Confidence Interval: Mean of C&amp;K Metrics for Apache Tomcat 5.5.9.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Lower Limit</th>
<th>Upper Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIT</td>
<td>0.84</td>
<td>1.08</td>
</tr>
<tr>
<td>NOC</td>
<td>0.29</td>
<td>0.51</td>
</tr>
<tr>
<td>WMC</td>
<td>10.99</td>
<td>12.97</td>
</tr>
<tr>
<td>CBO</td>
<td>7.43</td>
<td>8.95</td>
</tr>
<tr>
<td>RFC</td>
<td>22.09</td>
<td>27.17</td>
</tr>
<tr>
<td>LCOM</td>
<td>54.01</td>
<td>84.84</td>
</tr>
</tbody>
</table>

References
Key Success Factors in Software Engineering

ISBSG Software Project Repository & ISO 9126:
An Opportunity for Quality Benchmarking

Laila Cheikhi, Alain Abran, and Luigi Buglione

The International Software Benchmarking Standards Group (ISBSG) provides the Software Engineering community with a repository of project data which, up to now, have been used mostly for benchmarking and for estimating project effort. The 2005 version of the ISBSG repository includes data on more than 3,000 projects from various countries, sized with different functional size measurement methods and including a number of quality-related variables. ISO/IEC 9126 (International Organization for Standardization/International Electrotechnical Commission) is a series of ISO documents for the evaluation of the quality of software products: it proposes three quality models (internal quality, external quality and quality in use), together with the ISO taxonomy of quality characteristics and subcharacteristics. ISO 9126 also includes an inventory of over two hundred measures of the quality subcharacteristics. The goal of this paper is to identify whether or not the current ISBSG repository can be of use for benchmarking software product quality on the basis of ISO 9126.

Keywords: Benchmarking, ISBSG, ISO/IEC 9126, Project Data, Quality Models.

1 Introduction

Over the past few years, there has been a growing interest in benchmarking both organizational and project performances across the Information and Communications Technology (ICT) sector. Some of the issues to deal with when looking at the feasibility of a benchmarking exercise are: the availability and size of benchmarking repositories and, obviously, the quality of such data. In the ’90s, the International Software Benchmarking Standards Group (ISBSG, <http://www.isbsg.org>) was set up to provide a worldwide repository of software projects, all measured with a functional size measurement method. The 2005 version of ISBSG repository [1] contains information on over 3,000 projects and provides data useful for multiple purposes, including comparison of project productivity and building effort estimation models. Such productivity and estimation models can be used to improve overall organizational capabilities in terms of project planning and monitoring [2].

In the pursuit of improved estimation and data analysis capability, repositories such as the ISBSG can help the Software Engineering community to better understand some of the cause-effect relationships by investigating a number of the variables available in the repository fields, as well as trying to figure out which ones contribute the most to the achievement of certain goals (i.e. increased productivity, shorter time-to-market, etc.).

Typically, the ISBSG, as well as other project data repositories, contains a number of descriptive variables, as well as quantitative data from which a number of ratios can be derived. While many of these variables focus on productivity-related issues from multiple viewpoints (project managers, designers, test managers, etc.), what are the data fields that can be of use for quality planning and control? To investigate the availability, as well as the coverage, of quality-related data in the ISBSG, the ISO 9126 series (International...
Key Success Factors in Software Engineering

Section 2 provides an overview of the quality models of ISO 9126-1, and Section 3 describes the ISBSG organization and gives an internal view of its repository of project data through an analysis of its data collection questionnaire. Section 4 presents a comparative analysis of the ISBSG questionnaire and ISO 9126. Finally, opportunities for using ISO 9126 and the ISBSG repository for benchmarking software product quality are identified in Section 5.

2 ISO 9126 – An Overview
The ISO 9126 series of documents consists of four parts under the general title “Information Technology – Software Product Quality”. The first part (ISO 9126-1) specifies the ISO software product quality model. The other three parts provide an inventory of candidate “metrics” that can be used to evaluate the characteristics and subcharacteristics of the quality model. The software product is defined in ISO 9126 as “the set of computer programs, procedures, and possibly associated documentation and data. Products include intermediate products, and products intended for users such as developers and maintainers” [3].

The ISO 9126-1 quality model is defined as “a framework which explains the relationship between different approaches to quality” [3], and distinguishes three views of software quality: internal quality, external quality and quality in use (Figure 1):

- **Internal quality** corresponds to the “totality of the characteristics of the software product from an internal viewpoint”, which can be achieved by measuring the internal properties of the software product without executing it.
- **External quality** corresponds to the “totality of characteristics of the software product from an external viewpoint”, which means that the quality of the software product can be evaluated during its execution by measuring its external properties.
- **Quality in use** represents the “user’s view of the quality of the software product when it is used in a specific environment and a specific context of use”. It corresponds to the use of the software during the operation and maintenance phases, and is not related to its intrinsic properties.

The set of ISO 9126 quality views in Figure 1 is based on the belief that internal quality has an impact on external quality, which in turn has an impact on quality in use. Therefore, the achievement of quality in use depends to some extent on the achievement of external quality, which in turn depends on the achievement of the internal quality of the software product itself.

The internal and external quality models share the same hierarchical structure, with two levels. The first level has six characteristics, which are broken down into 27 subcharacteristics (Figure 2) in the second level. A set of internal and external measures approved by the ISO to specify and quantitatively assess these quality characteristics is provided in ISO 9126 Technical Reports, Part 2 [4] and Part 3 [5]. The quality in use model has only one level, and includes four characteristics (Figure 3) with a set of measures provided in ISO 9126 Technical Report, Part 4 [6]. These technical reports are not intended to give an exhaustive set of measures for all the characteristics; they provide only those measures for which there is a consensus within the ISO.

In Software Engineering, it is expected that a proper and exhaustive identification of project specifications and goals early in the development phases decreases the risk of rework and delay and of being over budget. Similarly, it is expected that evaluating the internal and external quality of the software product before delivery provides an opportunity to correct errors, to implement required changes and to decrease the risk of expensive rework and unforeseeable costs.

3 Overview of The ISBSG
3.1 ISBSG Organization
The ISBSG is a not-for-profit organization created in 1994 “to develop the profession of software measurement by establishing a common vocabulary and understanding of terms” [7]. It groups together national associations on

---

Figure 1: Quality along the Software Life Cycle (ISO/IEC 9126-1) [3].
software measurement, currently representing 13 different countries. The ISBSG software project repository provides "software development practitioners with industry output standards against which they may compare their aggregated or individual projects, and real data of international software development that can be analyzed to help improve the management of IT resources by both business and government" [8].

To achieve these goals, the ISBSG makes available to the public a questionnaire to collect data about projects, including software functional size measured with any of the measurement standards recognized by the ISO (COSMIC-FFP functional size – ISO 19761, etc.). Thereafter, the ISBSG assembles this data in a repository and provides a sample of the data fields to practitioners and researchers in an Excel file, referred to hereafter as the ISBSG MS-Excel data extract (Figure 4).

3.2 ISBSG Internal View

The internal view of the ISBSG data repository corresponds closely to their data collection questionnaire, with some additional fields added by their repository manager. The data collection questionnaire includes a large amount of information about project staffing, effort by phase, development methods and techniques, etc. Moreover, the ISBSG provides a glossary of terms and measures [7] to facilitate understanding of the questionnaire, to assist users at the time they collect data and to standardize the data-gathering process.

The ISBSG data collection questionnaire includes 7 sections subdivided into several subsections (Figure 5):

A. **Submitter Information**: in this section, information is collected about the organization and the people filling out the questionnaire. Such information is kept confidential by the ISBSG.

B. **Project Process**: in this section, information is collected about the project process. The ISBSG provides well-defined terms to describe this section, offers a simple structure to gather data and allows precise comparisons to be made among projects. The information collected here is structured along the various phases of a software life cycle (SLC): planning, specification, design, programming, test and implementation.

C. **Technology**: in this section, information is collected about the tools used for developing and carrying out the project. For each stage of the software project, the ISBSG questionnaire proposes a list of tools.

D. **People and Work Effort**: three groups of people are considered in this section: development team, customers and end-users, and IT operations. Collected here is information about the various people working on the project, their roles and their expertise, and the effort expend for each SLC phase.

E. **Product**: in this section, information is collected

![Figure 2: Quality Model for External and Internal Quality (ISO/IEC 9126-1) [3].](image-url)

![Figure 3: Quality model for Quality In Use (ISO/IEC 9126-1) [3].](image-url)
Key Success Factors in Software Engineering

about the software product itself (e.g. product description, application type and deployment platform, such as client/server, etc.).

F. COSMIC Project Functional Size: in this section, information is collected about the functional size of the project and a few other variables related to the context of the functional size measurement. The ISBSG COSMIC questionnaire includes, for example, tables for collecting quantitative information about data movements (ENTRIES, EXITS, WRITES and READS) for each type of project: new development or redevelopement software, or enhancement software. Again, some information is collected about the expertise of the software functional size measurer.

G. Project Completion: this last section of the questionnaire provides an overall picture of the project, including project duration, defects, number of lines of code, user satisfaction and project costs, including cost validation.

This data collection questionnaire consists of 134 questions within the seven data groups; some of these questions may contain a number of sub-questions. Therefore, the

---

**Figure 4:** ISBSG Organization.

**Figure 5:** Structure of The ISBSG Data Collection Questionnaire.
number of data fields collected by the questionnaire is actually larger; while a few of these data fields are mandatory, most are optional.

4 Comparison of ISBSG and ISO 9126

The ISO 9126 quality model refers to three types of quality shown in Figure 1 (internal quality, external quality and quality in use) covering the various phases of the SLC. The ISBSG questionnaire for project data collection is available on their Web site, and has been used as the key input to analyze whether or not the ISBSG repository contains the appropriate information for using ISO models of software quality.

The comparison of the ISBSG questionnaire phases with the ISO 9126 quality models is presented in Table 1; it can be observed, in particular, that there is full coverage at the high level of comparison.

At a more detailed level, the ISBSG questionnaire collects information for most of the SLC phases, and the ISBSG questions can be mapped to the three ISO 9126 views of software quality (internal, external and quality in use). More specifically (Table 2):

- the Project Process section can be mapped to two of the three views: internal and external;
- the Project Completion section can be mapped to the quality in use view;
- the remainder of the sections, Technology, People and Work Effort, Product and COSMIC Project Functional Size, cannot be mapped directly to any of the three ISO 9126 quality models (Table 2). However, information from these other sections can be of use to normalize the information about quality (for instance, using functional size) or to analyze causal relationships with various variables.

The ISBSG data collection questionnaire collects information related to the project entity with its various characteristics, whereas standard ISO 9126-1 focuses on part of this entity, that is, the quality of the software product. Since standard ISO/IEC 9126 is taken as a tool for the analysis, the study conducted in this paper is restricted to the quality of the software product within the software projects in the ISBSG repository. A set of quality related data fields are presented in Table 3.

<table>
<thead>
<tr>
<th>ISBSG Data Collection Questionnaire</th>
<th>ISO 9126</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Internal Quality</strong></td>
<td><strong>External Quality</strong></td>
</tr>
<tr>
<td>Specification</td>
<td>Build or Programming (Unit Testing)</td>
</tr>
<tr>
<td>Design</td>
<td>Test</td>
</tr>
<tr>
<td>Build or Programming (Review/Inspection)</td>
<td>Implementation or Installation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Comparison of The ISBSG Questionnaire Phases with The ISO 9126 Quality Models.

In this paper, we have investigated the availability, as well as the coverage, of quality-related data in the ISBSG repository using the ISO 9126 series on software product quality measurement as the baseline reference for this analysis. The ISBSG repository has been compared with the three ISO models of software quality (internal quality, external quality and quality in use) using the ISBSG data collection questionnaire. This comparison indicates that, based on the definition of its data fields, the ISBSG repository contains many of the fields required to collect information on the three views of the ISO models of software quality. The results of this analysis documented here can be very useful:

- to industry, in analyzing the availability of quality-related information in the ISBSG repository;
- to industry and researchers looking for information to analyze and implement ISO models of software quality;
- to the ISBSG organization, in studying how to improve the alignment of their software quality-related data collection standards to the ISO software quality models;
- to the ISBSG organization, in promoting their re-
<table>
<thead>
<tr>
<th>ISBSG Questionnaire Sections</th>
<th>Quality Views</th>
<th>Internal Quality</th>
<th>External Quality</th>
<th>Quality in Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Submitter Information</strong></td>
<td>General information</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Project Process</strong></td>
<td>Process infrastructure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Planning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specification</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Design</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Build or Programming</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implementation/installation</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Project management and monitoring</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Technology</strong></td>
<td>General information</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>People and Work Effort</strong></td>
<td>Development Team</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Customers and End Users</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IT Operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Work Effort validation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Product</strong></td>
<td>General Information</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>COSMIC Project Functional Size</strong></td>
<td>New development or redevelopment software size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enhancement software size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Context of the functional size measurement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Experience of the functional counter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Project Completion</strong></td>
<td>General Information</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>User satisfaction survey</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Project costs</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Cost Validation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: ISO 9126 Quality Views in The ISBSG Data Collection Questionnaire.

This study has been limited to the comparison with ISO 9126-1. Further work is being carried out to analyze the alignment of the ISBSG quality-related data fields to the inventory of over 200 measures proposed in Technical Reports 2 to 4 of the ISO 9126 series on software quality evaluation. The next steps taken will involve a detailed comparison of the ISBSG repository and the ISO 9162-1 models of software quality, through a high-level mapping of the ISBSG questionnaire to the three ISO quality views (internal, external and quality in use), followed by a detail-level mapping for each view.

References

Table 3: ISBSG Quality-related Data Fields.

<table>
<thead>
<tr>
<th>Categories</th>
<th>Phases</th>
<th>Quality-Related Data Fields Collected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Process</td>
<td>Specification</td>
<td>- Number of defects recorded in the documents and other work products of this phase</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Resolution/rework effort</td>
</tr>
<tr>
<td></td>
<td>Design</td>
<td>- Number of defects recorded during design</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Resolution/Rework effort</td>
</tr>
<tr>
<td></td>
<td>Build or Programming</td>
<td>Number of change requests made during design</td>
</tr>
<tr>
<td></td>
<td>Test</td>
<td>- Number of defects (minor, major, extreme or total) recorded and resolved during this activity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Resolution/Rework effort</td>
</tr>
<tr>
<td></td>
<td>Implementation or Installation</td>
<td>Number of change requests made during build</td>
</tr>
<tr>
<td></td>
<td>General Information</td>
<td>Number of change requests made during testing</td>
</tr>
<tr>
<td></td>
<td>User Satisfaction Survey</td>
<td>Number of change requests made during implementation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Number of defects recorded during the first month of the software’s operation (minor, major, extreme or total)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The lines of code generated by this project.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The percentage of these lines of code that are not program statements</td>
</tr>
<tr>
<td></td>
<td>User Satisfaction Survey</td>
<td>- Did the project meet the stated objectives?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Did the software meet business requirements?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Quality expectations for the software?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Quality expectations for the user documentation?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Ease-of-use requirements for the software?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Was sufficient training or explanation given?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Schedule for planning &amp; specification?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Schedule for design, build, test &amp; implement</td>
</tr>
</tbody>
</table>

52 UPGRADE Vol. VII, No. 1, February 2006
Requirements Elicitation in Web Engineering

Stephanos Mavromoustakos and Katerina Papanikolaou

Poor and incomplete design causes web systems to fail to meet users’ expectations and business’ goals. A major cause of failure of these systems is the non-inclusion of key success factors. These factors are incorporated into a cross-relational structure comprised of three axons: socio-cultural characteristics, user requirements and application domain. The present work identifies key success factors in web engineering and proposes a requirements engineering approach for eliciting requirements. The purpose of this method is to provide a simple way for analysts to identify these hidden requirements which otherwise could be missed or given little attention.

Keywords: Key Success Factors, Socio-Cultural Factors, Web Engineering.

1 Introduction

Researchers have recently demonstrated the importance of human, social and cultural (HSC) factors in web engineering, proving that these constitute significant factors which if ignored will lead to poor system design and a departure from business goals. Examples can be found in Fraser and Zarkada-Fraser [1], who have illustrated that ethnic groups follow different decision-making pathways in determining the web site they prefer to buy from, and that significant differences exist between cultures. Furthermore, Olsina, et al. [2] examined the quality of six academic operational sites to understand the level of fulfillment of essential quality characteristics, given a set of functional and non-functional user requirements from the viewpoint of students. The latter work proposed a quality requirement tree specifically for academic domains, classifying the elements that might be part of a quantitative evaluation, comparison and ranking process. In addition, Mavromoustakos et al. [3] examined the importance of human and socio-cultural factors in engineering quality e-learning applications by developing such a system. Papanikolaou and Mavromoustakos [4] have also identified human and socio-cultural factors in the context of mobile learning applications.

While there is a plethora of research works in web site engineering, there is lack of methods for revealing HSC factors and user requirements, which otherwise stay well hidden within the working environment analyzed. The risk of missing the requirements resulting from these factors leads us to propose a novel method to uncover and analyze HSC factors and user requirements, as well as to translate them into system requirements to provide a quality web application. Taking into consideration the importance of immediacy in deploying web applications, an oriented form of ethnography analysis is introduced, which can be conducted in a non-time consuming manner to identify requirements sourcing from HSC factors, based on a certain informational profile developed via focus questions.

The structure of this paper is as follows: Section 2 describes the socio-cultural characteristics, user requirements and application domain axons and identifies all the key success factors for web engineering under each axon. Section 3 proposes a requirements elicitation method for these critical factors and introduces a special form of ethnography analysis as an information gathering method for web applications. Section 4 sums up the findings of the paper and provides some concluding remarks.

2 Key Success Factors in Web Engineering

In this section, we identify all the key success factors in web engineering. To better visualize and understand these factors and the specific requirements for developing a quality web application we utilize a 3-axon model indicating the

Authors

Stephanos M. Mavromoustakos currently is Assistant Professor at the School of Computer Science and Engineering at Cyprus College. Previously, he worked as a Managing Director for a reputable software development company. He received his PhD from University of Cyprus and his MSc, MBA and BSc degrees from Nova Southeastern University, USA. His research interests include software and web engineering, human-computer interaction, e-learning, e-commerce and information security. He has published several articles in refereed journals and international conferences. He participated in many national and EU funded research projects. <mstefanos@cycollege.ac.cy>

Katerina Papanikolaou is Assistant Professor of Computer Science, Acting Dean of the School of Computer Science and Engineering, Cyprus College. Undergraduate studies at Imperial College, London Dept. of Physics and postgraduate studies at University College London, Dept. Electronic and Electrical Engineering (BSc in Physics 1993, PhD in 2000) She joined the Dept. of Computer Science of Cyprus College in 1999. She was the Computer Engineering Coordinator 2001-2002 and Dept. chairperson 2002-2004. Currently, she is the Acting Dean of the School of Computer Science and Engineering. Her research interests include broadband and multimedia communications, mobile networks routing, communications protocols, web engineering, e-commerce and e-learning applications. Partner and coordinator in several national and European projects. Presented her work in several referred national and international journals. <apapanik@cycollege.ac.cy>
interrelationship. The three axons are: The socio-cultural characteristics, the user requirements, and the application domain axon (Figure 1). Each axon includes certain components, which are directly connected and interrelated. The three axons are also interdependent, allowing the sharing of same, similar, or different characteristics amongst each other (Table 1). A description of each of the axons is presented now.

2.1 Socio-Cultural Characteristics

A web application must be tailored-made for each country or region of countries. In the requirements analysis phase the emphasis should be put on the range of countries which the web application will target and give special attention to the specific characteristics of the region for successful system development. These characteristics include:

Demographics
It is well known that human behavior varies according to gender and age. Therefore, these issues can significantly affect system design. The web engineer or project manager must specify and design the web application based on the targeted population. In addition, when introducing new products and services to a region it is important to have access to the various channels of distribution for achieving short-term and long-term organizational goals.

Social Characteristics
The analyst/developer must examine the educational system, the literacy level, as well as the languages spoken within the population, in order for the web application to be designed in such a way that will accommodate divergent features. Religion plays a significant role in politics, culture and economy in certain countries. Thus, the analyst must investigate whether religion affects the system design and to what degree.

Legal Characteristics
The political system and legislation among countries vary; therefore one must investigate political stability and all the relevant laws prior to the development of an e-commerce application. National and international laws must be analyzed to guide the system towards alignment and compliance upon full operation.

Technical Characteristics
Identifying the technology level of each targeted country will help the web engineer to decide on the type of technology and resources to use. Countries with advanced technologies and high web usage are excellent candidates for most types of web applications. On the other hand, countries new to the Internet arena will need time to adapt to this challenging electronic environment before taking the risk of doing business on-line.

2.2 User Requirements

The User Requirements axon follows the general software quality standards as defined by ISO 9126 and the web
engineering guidelines proposed by Olsina [2]. Each component is decomposed into several features that must be separately addressed to fulfill these user needs:

**Usability**

Issues like understandability, learnability, friendliness, operability, playfulness and ethics are vital design factors that web engineers cannot afford to miss. The system must be implemented in such a way to allow for easy understanding of its functioning and behavior even by non-expert Internet users.

Aesthetics of user-interface, consistency and ease-of-use are attributes of easy-to-learn systems with rapid learning curve. web systems, by storing user profiles and taking into consideration human emotions, can provide messages which are tailored to the user, whether this is a welcome message or an order confirmation note, thus enhancing the friendliness of the system. Playfulness is a feature that should be examined to see whether the application requires this characteristic and, if so, to what extent. web systems must reflect useful knowledge which reflects human interactions and decisions.

**Functionality**

The system must include all the necessary features to accomplish the required task(s). Accuracy, suitability, compliance, interoperability and security are issues that must be investigated in designing a web application to ensure that the system will perform as it is expected to. The web application must have searching and retrieving capabilities, navigation and browsing features and application domain-related features [2].

**System Reliability**

Producing a reliable system involves understanding issues such as fault tolerance, crash frequency, recoverability and maturity. The system must maintain a specified level of performance in case of software faults with the minimum number of crashes possible. It also must have the ability to re-establish its level of performance.

A system must consistently produce the same results, and meet or even exceed users’ expectations. The web application must have correct link recognition, user input validation and recovery mechanisms.

<table>
<thead>
<tr>
<th>AXON</th>
<th>COMPONENT</th>
<th>DECOMPOSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socio-Cultural Characteristics</td>
<td>Demographics</td>
<td>Gender, age</td>
</tr>
<tr>
<td></td>
<td>Social characteristics</td>
<td>Language, literacy, religion</td>
</tr>
<tr>
<td></td>
<td>Legal characteristics</td>
<td>International and domestic laws</td>
</tr>
<tr>
<td></td>
<td>Technical characteristics</td>
<td>web access, type of technology</td>
</tr>
<tr>
<td>User Requirements</td>
<td>Usability</td>
<td>Understandability, learnability, operability, playfulness</td>
</tr>
<tr>
<td></td>
<td>Functionality</td>
<td>Suitability, accuracy, compliance, interoperability, security</td>
</tr>
<tr>
<td></td>
<td>Reliability</td>
<td>Fault tolerance, crash frequency, recoverability, maturity</td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>Time behavior, resource behavior</td>
</tr>
<tr>
<td></td>
<td>Timeliness</td>
<td>Business goals</td>
</tr>
<tr>
<td></td>
<td>Maintainability</td>
<td>Analyzability, changeability, stability, testability</td>
</tr>
<tr>
<td>Application Domain</td>
<td>Informational</td>
<td>Online newspapers, electronic books, newsletters</td>
</tr>
<tr>
<td></td>
<td>Interactive</td>
<td>Registration forms, customized presentations, online games</td>
</tr>
<tr>
<td></td>
<td>E-commerce/Transactional</td>
<td>Electronic shopping, online banking</td>
</tr>
<tr>
<td></td>
<td>Workflow</td>
<td>Online planning and scheduling systems, status monitoring</td>
</tr>
<tr>
<td></td>
<td>Collaborative work</td>
<td>Distributed authoring systems, collaborative design tools</td>
</tr>
<tr>
<td></td>
<td>environments</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Online communities</td>
<td>Chat groups, online auctions</td>
</tr>
<tr>
<td></td>
<td>marketplaces</td>
<td></td>
</tr>
<tr>
<td></td>
<td>web portals</td>
<td>Online intermediaries, electronic shopping malls</td>
</tr>
</tbody>
</table>

Table 1. Axon Categorization of the SpiderWeb Model
Key Success Factors in Software Engineering

**Efficiency**

A web system’s goal (especially an e-commerce site) is usually to increase productivity, decrease costs, or a combination of both. Users expect the system to run in an efficient manner in order to support their goals. The system’s response-time performance, as well as page and graphics generation speed, must be high enough to satisfy user demands. Fast access to information must be examined also throughout the system life to ensure that users’ requirements are continuously met whilst the system remains competitive and useful.

**Maintainability**

Some crucial features related to maintaining a web application is its analysability, changeability, stability, and testability. The primary target here is to collect data that will assist designers in conceiving the overall system in the best architectural and modular form from a future maintenance point of view. With the rapid technological changes especially in the area of web engineering, as well as the rigorous users’ requirements for continuous web-site updates, easy system modifications and enhancements, both in content and in the way this content is presented, are also success factors for the development and improvement of a web application. Another important area the researcher must concentrate on is the timeliness of the content (i.e. the information processed within the system), the functionality (i.e. the services offered by the system) and the business targets (i.e. the business goals using the system) the e-commerce system must exhibit. Timeliness is examined through a cultural prism aiming at identifying certain human, social, and organizational needs in all three of its coordinates, as most of the applications exhibiting a high rate of change often depend highly on the ethos and customs of different people in different countries (i.e. electronic commerce systems).

### 2.3 Application Domain

The web engineer should investigate users’ satisfaction on existing similar web applications and their expectations on visiting the site. Depending on the type of application (see Table 1) the web engineer should also identify the driving factors users visit, purchase and use the site. Emphasis should also be given on users concerns, feelings, trust and readiness of using a web application. These can be identified through a requirements elicitation process which is proposed in the next section.

### 3 Requirements Elicitation Process

The description and brief analysis of the axon components presented in the previous section aimed primarily at providing the basic key concepts that developers must utilize to collect proper system requirements. These concepts will be used as guidelines for the significant process of gathering critical information that may affect the functional and non-functional behavior of the system under development. We propose the use of an oriented form of ethnography analysis conducted in a small-scale time-wise manner for collecting and analyzing information for the three axons described before.

Ethnography originates from anthropology where it was primarily used in sociological and anthropological research as an observational analysis technique, during which anthropologists study primitive cultures [5]. Today, this form

<table>
<thead>
<tr>
<th>Country Characteristics</th>
<th>Focus Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demographics</td>
<td>What is the gender and age of the targeted population?</td>
</tr>
<tr>
<td></td>
<td>What are the channels of distribution?</td>
</tr>
<tr>
<td></td>
<td>Are the neighboring countries open for electronic trade of goods and services?</td>
</tr>
<tr>
<td>Social Characteristics</td>
<td>What are the main languages spoken in the region?</td>
</tr>
<tr>
<td></td>
<td>What is the religion of the targeted population?</td>
</tr>
<tr>
<td></td>
<td>What is the literacy percentage grouped by gender and age?</td>
</tr>
<tr>
<td></td>
<td>What is the level of efficiency of the educational system with respect to the web?</td>
</tr>
<tr>
<td>Legal</td>
<td>Is there political stability in the area?</td>
</tr>
<tr>
<td></td>
<td>Are there any laws that prohibit the electronic sale of certain goods?</td>
</tr>
<tr>
<td>Technical</td>
<td>What is the percentage of the targeted population with web access, by gender and age?</td>
</tr>
<tr>
<td></td>
<td>What is the web access rate of increase?</td>
</tr>
<tr>
<td></td>
<td>What is the average transmission speed to browse the Internet?</td>
</tr>
</tbody>
</table>

Table 2: Focus Questions for Collecting Socio-Cultural Characteristics Factors.
of analysis constitutes a valuable tool in the hands of software engineers by utilizing techniques, such as observations, interviews, video analyses, questionnaires and other methods, for collecting human and socio-cultural factors. In a design context, ethnography aims to provide an insight understanding of these factors to support the design of computer systems [6][7][8][9][10].

This approach offers great advantages in the system development process by investigating these key factors and exploring human activity and behavior that otherwise software engineers would have missed. Examples can be seen in several studies performed in a variety of settings, including underground control rooms [11], air traffic control [12], police [13], banking [14], film industry [5] and emergency medicine [15].

Bearing in mind that ethnography analysis is time consuming by nature and the immediacy constraint in deploying web applications [16], we propose a short-scale form of ethnography analysis, focusing on cognitive factors. Our proposition is based upon examining the existing working procedures of the client organization, either manual or computerized, together with the users’ behavior. Specifically, the working environment of the organization and its employees, as well as a group of customers currently doing business transactions with the organization are set as targeted population of the analysis, utilizing this shortened form of ethnography on the three axons of our model. The short-scale ethnography analysis may include observations, interviews, historical and empirical data, as well as questionnaires. Emphasis is given to focus questions produced in the form of questionnaires. These questions are distributed among the targeted group or are used as part of the interviewing process, and the answers are recorded, analyzed and evaluated. Data collection mechanisms, as well as the kind of information for analyzing each primary component in the axons of the proposed model, are defined in the 3-axon approach via a profile shell that web engineers must develop before requirements analysis starts. Each component is associated with suggested focus questions provided in Tables 2 through 4. It must be noted that these are a proposed set of key questions for the analyst to use as guidelines, but he may also enhance the set with other application-specific questions he may regard as equally essential for the application under development.

4 Conclusions

Human, social and cultural factors in web engineering are significant factors which if ignored will lead to poor system design and a departure from business goals. In this paper, we have identified the key success factors for developing a quality web application. These factors are classified into three main axons, namely socio-cultural factors, user requirements and application domain. The 3-axon approach is utilized to better understand their interrelationship. Each axon includes certain components, which are directly connected and interrelated. The three axons are also

<table>
<thead>
<tr>
<th>User Requirements</th>
<th>Focus Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>How do expert and non-expert Internet users understand the system?</td>
</tr>
<tr>
<td></td>
<td>Are easy-to-learn systems too complicated for expert users?</td>
</tr>
<tr>
<td></td>
<td>How do users perceive content layout and how does this affect user retention?</td>
</tr>
<tr>
<td></td>
<td>How does the system handle the conflicting requests for maximum or minimum playfulness?</td>
</tr>
<tr>
<td></td>
<td>How does the content layout (colors, menus, consistency) affect web usage?</td>
</tr>
<tr>
<td></td>
<td>What is the level of sensitivity in ethical issues among the targeted user group and how does this affect the way they interact with the web?</td>
</tr>
<tr>
<td></td>
<td>What is the level of trust for ensuring privacy?</td>
</tr>
<tr>
<td></td>
<td>How can on-line shopping be more entertaining than in-store shopping?</td>
</tr>
<tr>
<td>Functionality</td>
<td>How do users feel with registering to a web application is a prerequisite for accessing its content?</td>
</tr>
<tr>
<td></td>
<td>What is the required level of security of functions, for individuals to provide their credit card for on-line purchases?</td>
</tr>
<tr>
<td></td>
<td>What is the maximum bearable time for users to wait in search for information before dropping the site?</td>
</tr>
<tr>
<td>Maintainability</td>
<td>How often will users need to see content updates?</td>
</tr>
<tr>
<td></td>
<td>Are market conditions matured for such a system?</td>
</tr>
<tr>
<td></td>
<td>How do people accept system changes?</td>
</tr>
<tr>
<td>Reliability</td>
<td>What is the acceptable fault tolerance that will not drive away existing users?</td>
</tr>
<tr>
<td>Efficiency</td>
<td>At what degree users expect to decrease their costs? Can these expectations be met?</td>
</tr>
</tbody>
</table>

Table 3. Focus Questions for Collecting User Requirements.
interdependent, allowing the sharing of same, similar, or different characteristics among each other. The socio-cultural axon includes demographics, social, legal and technical characteristics of the country or region of countries where the application will be focused. The user requirements axon comprises of all the specific requirements related to usability, functionality, efficiency, reliability and maintainability. Finally, the application domain axon includes all the peculiarities and characteristics of the various types of web applications. In this paper, we also propose a requirements elicitation process for gathering these factors. This process is based on a special form of ethnography analysis which due to the immediacy constraint of developing web applications is performed in a short time-scale. Further work includes the development of a web engineering process incorporating all the key success factors for constructing quality web applications.

References

Focus Questions

<table>
<thead>
<tr>
<th>Focus Questions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Are users satisfied with the current e-commerce sites? What are their recommendations for improvement?</td>
<td></td>
</tr>
<tr>
<td>What do users expect to find, shopping in an e-commerce application versus shopping in a traditional store?</td>
<td></td>
</tr>
<tr>
<td>How does a user behavior change when using long versus short registration forms?</td>
<td></td>
</tr>
<tr>
<td>Are users ready for e-commerce, both in b2b and b2c?</td>
<td></td>
</tr>
<tr>
<td>What are the users’ feelings and trust on doing business on-line?</td>
<td></td>
</tr>
<tr>
<td>What are the users’ concerns and doubts on security, product delivery, efficiency, and company legitimacy?</td>
<td></td>
</tr>
<tr>
<td>What types of auctions users are accustomed to?</td>
<td></td>
</tr>
<tr>
<td>How easily users are affected by outside factors in their shopping decisions?</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Focus Questions for Collecting Application Domain Factors (e.g. e-Commerce/Transactional System).
Software Engineering (SE) is a well established and recognized discipline with a significant influence on the software development world and on Information Technology (IT) in general. Many papers have highlighted SE as the basis for a differentiated set of professional profiles in the field of computing, and have stressed the influence of Human Resources (HR) on software productivity, but neither the standards nor the many empirical studies so far produced have provided an in-depth and wide-ranging analysis of the various specialization roles involved in a project. In this paper we take the connection between productivity and staff profiles and training as a given, and present an overview of the facts and figures concerning the definition of roles and positions in software development.

Keywords: Human Resources, Job Profiles, Productivity, Software Engineering.

1 Introduction

Companies tend to say that "human resources are our main asset". This is especially true in software development where cost estimation models and methods recognize people as the main resource and cost: in fact, estimation models are based on the estimation of human effort (work hours or months) to calculate the cost of project [1]. "Software engineer" is of course recognized as a separated and clearly defined IT (Information technology) job profile [2], but the people involved in software development projects play a great variety of roles and it is normal to differentiate between a number of different positions. We therefore need to study the different characteristics (relevant to each position or role) that influence people’s ability to contribute to productivity and quality in software development:

- Academic education and specific training courses: our studies of IT job applicant requirements in Spain [3] clearly show the rapid evolution of and changes in the required knowledge of technical matters, and the increasing need to demonstrate personal skills such as the ability to work in teams, communication skills, proactive attitude, etc..
- Specific experience and skills related to technical environments, programming languages, data base systems, etc.
- Deployment of best practices for personal development in software projects that influence productivity and quality: e.g. PSP [4].

Neither can we ignore the strong influence exerted by motivation and a culture of quality, the spirit of excellence, etc. which are especially dependent on an ethical attitude such as is set out in the IEEE code of ethics for software engineers (see <http://www.computer.org/certification/ethics.htm>).

Although the study of people’s work is in itself an exciting discipline, we need to take a special interest in analysing the effect of a number of factors related to Human Resources (HR) on software development productivity. If people are the main cost driver, it is important to study all the relationships between developers and productivity, quality, or any other commonly established goal for software projects. Productivity in particular is a much debated subject because software development is regularly blamed for low productivity. In [5] for example, various experts comment on a report published in Business Week saying that software productivity has decreased by 0.9 percent a year in the United States (the lowest productivity of all other sectors or activities). Most experts point to the impact of technological advances, new and more powerful tools, and automation, but only in very few cases do they mention the
problems related to process improvement and the need for caution when incorporating new tools into organizations. And only one of them makes mention of the problem of communication barriers and the fact that the rate of adoption of new tools (and therefore processes) by developers is very low1. There were no references at all to HR, training, education, motivation, or experience.

2 HR and Productivity

Various papers have attempted to formalize the relationship between people and productivity by addressing such issues as the skill and experience of developers. The well-known COCOMO [6] and COCOMO II models [7] include the factors related to HR shown in Table 1.

Two facts become apparent when analysing this model (leaving aside any discussion regarding the accuracy of the productivity influence values assigned to each factor):

- The negative influence of some factors (i.e. the low skills of analysts or programmers, or their little experience in the target environment) is greater than the corresponding positive influence on costs. For example, bad quality analysts may cause a 42% rise in costs over the nominal/standard effort but good quality analysts only reduce costs by 29%.

- The guidelines used to rank personnel are based on subjective methods where criteria include percentile evaluation (e.g. analyst capability is ranked as very low if it is in the 15th percentile according to a subjective evaluation of different skills) and a number of ranges for experience (e.g. less than 2 months counts as very low for experience related cost drivers).

Another interesting example of the quantification of the influence of several factors in productivity is presented in [8] (see Table 1). We can see that the influence of factors is generally greater when there is a negative influence (e.g. inexperienced developers: -87%) than when that influence is positive (e.g. experienced developers: +55%). Of course, many other factors closely related to HR (not only skill and experience) have a real influence on productivity: e.g. office ergonomics, motivation/morale, etc. (See Table 2.)

An important aid to understanding the relationships between the different factors affecting software development projects is Abdel-Hamid’s microworld-based software development model [9]. This is a good tool for simulating the real behaviour of development projects. It was developed on the basis of interviews with software project managers in five major organizations and an extensive database of empirical findings from relevant literature. It includes four main subsystems, one of which is devoted to human-resource management, dealing with hiring, assimilation and transfer of people. This subsystem also breaks the workforce down into various employee categories (for example: newly hired, experienced workers). It also reflects the fact that the training process to assimilate new employees causes veterans’ productivity to decline, as they are usually the ones to train newcomers. Other factors such as workforce stability and size tend to evolve as the project completion date nears and the project progresses. Such parameters as hiring rate, turnover rate, and workforce experience mix influence the project team structure which, in turn, has an effect on such aspects as potential productivity, software development rate, and project losses. However, the full model of Abdel-Hamid has more than 100 causal links between the different elements of the development world.

This model assumes the following equation [10]:

Productivity = potential productivity - losses due to faulty processes

Potential productivity refers to when an individual or group makes the best use of their available resources and is a function of the nature of the task and the team’s resources. Average productivity is a weighted average of two nominal productivity rates; one for experienced staff and one for newly acquired staff. There are also problems such as motivation (development and promotion, salary, responsibility,

### Table 1: People-related Cost Drivers in COCOMO Models.

<table>
<thead>
<tr>
<th>Cost driver</th>
<th>Very low</th>
<th>Low</th>
<th>Nominal</th>
<th>High</th>
<th>Very high</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyst capability</td>
<td>1.42</td>
<td>1.19</td>
<td>1.00</td>
<td>0.85</td>
<td>0.71</td>
</tr>
<tr>
<td>Programmer capability</td>
<td>1.34</td>
<td>1.15</td>
<td>1.00</td>
<td>0.88</td>
<td>0.76</td>
</tr>
<tr>
<td>Platform experience</td>
<td>1.19</td>
<td>1.09</td>
<td>1.00</td>
<td>0.91</td>
<td>0.85</td>
</tr>
<tr>
<td>Application experience</td>
<td>1.22</td>
<td>1.10</td>
<td>1.00</td>
<td>0.88</td>
<td>0.81</td>
</tr>
<tr>
<td>Language and tools experience</td>
<td>1.19</td>
<td>1.09</td>
<td>1.00</td>
<td>0.91</td>
<td>0.85</td>
</tr>
<tr>
<td>Staff continuity</td>
<td>1.29</td>
<td>1.12</td>
<td>1.00</td>
<td>0.90</td>
<td>0.81</td>
</tr>
</tbody>
</table>

1 Three common factors were detected: the lack of any industry-wide standard definition for software productivity, the increasing complexity of software applications, and the need for more formalized processes in the industry as a whole.

2 "Analysts are people who work on requirements, high-level design and detailed design" [7]. Evaluation criteria include analysis and design ability, thoroughness and efficiency, communication and cooperation skills (experience is excluded).

3 The evaluation should be based on the capability of the team rather than that of the individuals. Criteria are similar to those applied to analysts.

4 Based on turnover rate: >48%/year counts as very low continuity, <3%/year counts as very high.
recognition, etc.) and communication overhead.

We should remember that each communication link requires effort and time; in fact, at least 30% of programmers’ time is devoted to working alone [11][12] and a number of attempts have been made to devise better metrics for communication [13].

Several conclusions have emerged as consequence of a great deal of study and debate on this subject. For example, the inclusion of people late in a project is often disruptive [14]. The new people need to learn the system, and their teachers are none other than those already doing the work, while new communication paths also need to be established.

As shown in [9], the learning curve of new environments has a strong influence on productivity. Several papers (with empirical data) have highlighted the different aspects that need to be addressed by people management [15][16].

Various and sometimes radical rules-of-thumb have emerged from some of these studies: e.g. it is better to fire an incompetent programmer than to bring in another one [17] or that a good programmer can be 5 to 10 times more productive than an average worker [16].

We can therefore see that HR factors have a real influence on productivity and that the various proposals mentioned above have attempted to quantify that influence. But we believe that a more detailed insight is needed to establish a solid basis for this type of study. There are a great many different job positions and roles involved in a software development project.

Each of them has a very different profile which needs to be analysed if any further studies on productivity are to make sense. In the following section we present an analysis of a number of references (process standards, methodologies, job classifications, etc.) in which these positions and roles are supposedly described.

### 3 Profiles, Positions, and Roles in Software Development

In this overview we will take a look at a number of viewpoints regarding descriptions and profiles of the jobs and roles involved in software development projects. As can be seen below, our overview covers classifications that have arisen from process standards, software methodology handbooks, and job categories and certifications. Our analysis of this selection of documents is focused on finding similarities and differences in the different worker profiles.

#### 3.1 Process Standards

In the discipline of software development, a number of standard process descriptions have been defined as a formal reference framework to describe activities during projects. The following are particularly worthy of mention: ISO/IEC 12207:1995 [18] establishes a framework for software life cycle (SLC) processes. It sets out in detail the processes, activities and tasks suggested for SLC and also describes how to tailor the standard for a specific project. IEEE Std 1074-1997 [19] provides a process for creating an SLC process model including detailed descriptions of suggested activities and processes.

There are practically no descriptions of the profiles of people involved in processes; only generic titles are used to refer to personnel profiles (such as acquirer, developer, user, etc.).

#### 3.2 Software Development Methodologies

There are many well known methodologies in professional software development. In this analysis, we have focused on two of the most widely used methodologies in Spain: the Unified Software Development Process [20] and Métrica 3 [21].

Although strictly speaking this is not a methodology, USDP (Unified Software Development Process) is closely related to the application of UML (Unified Modeling Language) notation [22]. It describes a use-case driven, architecture-centric, iterative, and incremental process. For each phase identified in the process, there is a set of activities and responsibilities that workers involved in the process should cover. In USDP, ten different roles are established:

<table>
<thead>
<tr>
<th>Factors</th>
<th>Positive impact (+%)</th>
<th>Negative impact (-%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Staff experience⁵</td>
<td>High +55%</td>
<td>Low –87%</td>
</tr>
<tr>
<td>Management experience</td>
<td>High +65%</td>
<td>Low –90%</td>
</tr>
<tr>
<td>Office</td>
<td>Ergonomics +15%</td>
<td>Crowded space –27%</td>
</tr>
<tr>
<td>Unpaid overtime</td>
<td>Yes +15%</td>
<td>No %</td>
</tr>
<tr>
<td>Work</td>
<td>Specialist +18%</td>
<td>Generalist –15%</td>
</tr>
<tr>
<td>Morale</td>
<td>High +7%</td>
<td>Low –6%</td>
</tr>
<tr>
<td>Organization</td>
<td>Hierarchical +5%</td>
<td>Matrix –8%</td>
</tr>
</tbody>
</table>

Table 2. Impact of Key Adjustment Factors on Productivity.

---

⁵ Experience is evaluated in terms of application type, programming language, and programming tools or environment.

⁶ It should be noted that the ISO 15504 [17] life cycle processes description is closely based on ISO 12207.

⁷ In each iteration, a mini-cascade cycle is followed by the following phases (workflows): Requirements capture, Analysis, Design, Implementation, Testing.
System Analyst, Use-Case Specifier, User-Interface Designer, Architect, Use-Case Engineer, Component Engineer, System Integrator, Test Designer (Test Engineering), Integration Tester, and System Tester.

The responsibilities of these roles evolve throughout the process, so a team member may take on different roles during a project. The project leader should know individual competences so that the best suited people are assigned to their appropriate roles.

Métrica v3 is the official Spanish methodology for national public administration contracts. It provides a framework for the systematic performance of activities involved in SLC. Métrica consider that SLC participants can be broken down into five generic professional profiles (Manager, Project Leader, Consultant, Analyst, Programmer). The description of each profile includes responsibilities and duties for each phase of SLC and details the tasks associated with them. Table 3 shows the relationship between participants in the development process and the profiles identified in Métrica 3.

If we concentrate on the main processes for IS development (feasibility study, analysis, design, construction, implementation and acceptance, maintenance) and on the main tasks associated with these processes, the participant catalogue could be reduced to the first twenty roles/positions.

Similarities can be seen between the two methodologies/process proposals: although they do not include exactly the same tasks, the development processes identified in [21] could be deemed to be more or less equivalent to the phases described in [20] (see Table 4).

We can therefore compare workers’ tasks in the two methodologies, according to the phases in which they take part. The results are shown in Table 5.

In terms of the processes in which the various categories participate, the similarity between the profiles of the two methods is quite marked in certain cases (Architect from USDP with Analyst in Métrica 3, System integrator from USDP with Programmer from Métrica 3). But there are also cases in which no direct match is possible, e.g., Métrica 3 Implementation team could be associated with any of three USDP workers (Test Designer - Test Engineering, Integration Tester, System Tester). And, of course, there are

<table>
<thead>
<tr>
<th>ID</th>
<th>Development participants</th>
<th>Project Leader</th>
<th>Consultant</th>
<th>Analyst</th>
<th>Programmer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Project leader</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Implementation manager</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Maintenance manager</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Operations manager</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Systems manager</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Security manager</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Communications specialist</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Systems technician</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Communications technician</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Analyst</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Database administrator</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Project team</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Architectural team</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Implementation team</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Operations team</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Systems team</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Technical support team</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Training and education team</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Programmer</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Systems operations team</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Software quality manager</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Consultant</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Computing consultant</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>IT Consultant</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>IS Consultant</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Software Quality Assurance Group</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Métrica 3 Participants and Profiles.

---

8 Documentation is available, in Spanish, at the Ministry of Public Administration’s website: <http://www.map.es/csi>.
9 Each phase description includes participants and their main characteristics.
10 In this analysis we do not include any profile for the position of executive manager, normally associated with company responsibilities and decision making.
many profiles with no equivalent profile on the other side.

### 3.3 Professionalism: Job Classification and Certification

The International Standard Classification of Occupations [23] is a general set of occupational definitions for all workforce occupations. ISCO classifies jobs in terms of the type of work to be performed. The basic criteria used to define the groups used by the system are the levels of "skill" and "skill specialization" required to carry out the tasks and duties of the jobs.

Table 6 presents this classification scheme together with the job titles related to the classification group for each IT job. Our research focused on job titles involving software development (other computing positions are shown in grey).

The Standard Occupational Classification SOC 2000 [24] is a general set of occupational definitions for all workforce occupations. It was first published in 1990 (SOC 90) and has been revised and updated up to the present SOC2000. It describes typical entry routes and associated qualifications for each group and proposes a list of related job titles. Table 7 shows a summary of the group connected to IT. As we are concerned only with software development, once again there are a number of groups or titles that are not relevant to this article.

Although not linked to any official standardization organization, Career Space [25] has earned its status as a European benchmark for the IT profession. It is a consortium of major ICT companies working in partnership with the European Commission that has developed generic skills profiles relevant to key ICT jobs that cover the main job areas for which the ICT industry is experiencing skills shortages. Each core profile describes the jobs involved, setting out the vision, role, and lifestyle associated with them as well as the specific technology areas and tasks associated with each job and the level of behavioural and technical skills required to carry out the profiled jobs. Three of these profiles relate closely to the Software Development process & Test Engineering). In Table 8 there is a list of 53 associated job titles.

Another interesting slant on job descriptions is to be found in certification schemes. EUCIP (European Certification of Informatics Professionals) [26] is a qualification scheme for IT professionals that arose out of a CEPIS (the Council of European Professional Informatics Societies) initiative. One of the goals of this project is to define an "industry-driven vocational structure and standards for the informatics profession". There are 22 professional certifications planned but only four of them are available as yet (see those in bold type in Table 9).

If we compare Tables 6 to 9 there are many similarities to be found but, generally speaking, there are a large number of different classifications profiles and job classifications that are not easy to match.

<table>
<thead>
<tr>
<th>Metr. Id.</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD 1</td>
<td>System Analyst</td>
</tr>
<tr>
<td>USD 2</td>
<td>Use-Case Specifier</td>
</tr>
<tr>
<td>USD 3</td>
<td>User-Interface Designer</td>
</tr>
<tr>
<td>USD 4</td>
<td>Architect</td>
</tr>
<tr>
<td>USD 5</td>
<td>Use-Case Engineer</td>
</tr>
<tr>
<td>USD 6</td>
<td>Component Engineer</td>
</tr>
<tr>
<td>USD 7</td>
<td>System Integrator</td>
</tr>
<tr>
<td>USD 8</td>
<td>Test Designer (Test Engineering)</td>
</tr>
<tr>
<td>USD 9</td>
<td>Integration Tester</td>
</tr>
<tr>
<td>USD 10</td>
<td>System Tester</td>
</tr>
<tr>
<td>Metr. 1</td>
<td>Project leader</td>
</tr>
<tr>
<td>Metr. 2</td>
<td>Implementation manager</td>
</tr>
<tr>
<td>Metr. 3</td>
<td>Maintenance manager</td>
</tr>
<tr>
<td>Metr. 4</td>
<td>Operations manager</td>
</tr>
<tr>
<td>Metr. 5</td>
<td>Systems manager</td>
</tr>
<tr>
<td>Metr. 6</td>
<td>Security manager</td>
</tr>
<tr>
<td>Metr. 7</td>
<td>Communications specialist</td>
</tr>
<tr>
<td>Metr. 8</td>
<td>System technician</td>
</tr>
<tr>
<td>Metr. 9</td>
<td>Communications technician</td>
</tr>
<tr>
<td>Metr. 10</td>
<td>Analyst</td>
</tr>
<tr>
<td>Metr. 11</td>
<td>Data base administrator</td>
</tr>
<tr>
<td>Metr. 12</td>
<td>Project team</td>
</tr>
<tr>
<td>Metr. 13</td>
<td>Architectural team</td>
</tr>
<tr>
<td>Metr. 14</td>
<td>Implementation team</td>
</tr>
<tr>
<td>Metr. 15</td>
<td>Operations team</td>
</tr>
<tr>
<td>Metr. 16</td>
<td>Systems team</td>
</tr>
<tr>
<td>Metr. 17</td>
<td>Technical support team</td>
</tr>
<tr>
<td>Metr. 18</td>
<td>Training and education team</td>
</tr>
<tr>
<td>Metr. 19</td>
<td>Programmer</td>
</tr>
<tr>
<td>Metr. 20</td>
<td>System operations team</td>
</tr>
</tbody>
</table>

Table 5: Participants in Each Phase: Métrica 3 and USDP.

11 Job titles related to Data Communications Engineering, Multimedia Design, ICT Management or IT Business Consultancy are not included.
Table 6: ISCO Job Titles.

<table>
<thead>
<tr>
<th>Job titles</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyst, communications/computers</td>
<td>COMPUTER SYSTEMS DESIGNERS AND ANALYSTS</td>
</tr>
<tr>
<td>Analyst, database/computers</td>
<td>COMPUTER SYSTEMS DESIGNERS AND ANALYSTS</td>
</tr>
<tr>
<td>Analyst, systems/computers</td>
<td>COMPUTER SYSTEMS DESIGNERS AND ANALYSTS</td>
</tr>
<tr>
<td>Database administrator</td>
<td>COMPUTER SYSTEMS DESIGNERS AND ANALYSTS</td>
</tr>
<tr>
<td>Designer, systems/computers</td>
<td>COMPUTER SYSTEMS DESIGNERS AND ANALYSTS</td>
</tr>
<tr>
<td>Engineer, computer systems</td>
<td>COMPUTER SYSTEMS DESIGNERS AND ANALYSTS</td>
</tr>
<tr>
<td>Programmer</td>
<td>COMPUTER PROGRAMMERS</td>
</tr>
<tr>
<td>Programmer, communications</td>
<td>COMPUTER PROGRAMMERS</td>
</tr>
<tr>
<td>Programmer, database</td>
<td>COMPUTER PROGRAMMERS</td>
</tr>
<tr>
<td>Engineer, computer applications</td>
<td>COMPUTING PROFESSIONALS NOT ELSEWHERE CLASSIFIED</td>
</tr>
<tr>
<td>Assistant, computer/programming</td>
<td>COMPUTER ASSISTANTS</td>
</tr>
<tr>
<td>Assistant, computer/systems analysis</td>
<td>COMPUTER ASSISTANTS</td>
</tr>
<tr>
<td>Assistant, computer/users' services</td>
<td>COMPUTER ASSISTANTS</td>
</tr>
<tr>
<td>Operator, computer peripheral equipment</td>
<td>COMPUTER EQUIPMENT OPERATORS</td>
</tr>
<tr>
<td>Operator, computer peripheral equipment/high-speed printer</td>
<td>COMPUTER EQUIPMENT OPERATORS</td>
</tr>
</tbody>
</table>

4 Conclusion

This initial study provides us with a general overview of the huge variety of job titles used to define people working in software development. It would be interesting to have a more detailed breakdown of these job titles under the umbrella of a formal standard classification for professional profiles. Better still, it would take into account up to date information from the software industry (supply and demand of IT professionals) in order to stay in touch with present day requirements and descriptions of profiles, roles, and job titles. By using mechanisms such as the RENTIC reports [27], we are engaged in a line of research to help determine the specific requirements for each position.

The RENTIC database includes data from more than 1,200 job offers and covers 187 different job titles. From this data, we have determined that, for example, the most important personal skills for analysts are teamwork, initiative/proactivity, and autonomy/independence. Academia is interested in obtaining an insight into the labour market outlook in order to offer degree or specialized courses for each specific area. Researchers at the Université du Québec à Montréal (Canada), ACM and the IEEE Computer Society have been working on identifying the Software Engineering elements that are most widely accepted by both academia and industry.

This project, called SWEBOK (Software Engineering Body of Knowledge) [28] was initiated in 1998, and in 2004 the IEEE Computer Society approved the 2004 edition of its guide to the Software Engineering Body of Knowledge. The outcome of this research has been used in Computing Curricula 2004 [29] and, naturally, in SE 2004 [30] (Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering).

There is a similar initiative in Europe that analyses the situation of ICT Practitioner Skills/Competence Frameworks in an effort to create a European ICT Skills Meta-Framework for the proposed European Qualifications Framework (EQF). This generic research does not review specific software development environment or software development methodologies and standards, so it may be interesting as a basis for further analysis.

In any event, as McConnell [32] said, "the benefits of creating a true profession of Software Engineering are compelling", so it is important to promote a clear definition of the profiles involved in the software development process.

---

12 Specific reports focused on requirements for each job offer in IT in the Spanish labour market carried out by Dr. L. Fernandez-Sanz since 1998.

13 Including equivalent titles.

Table 7. SOC 2000 Unit Groups and Job Titles Relevant to The Software Development Process.

<table>
<thead>
<tr>
<th>1136</th>
<th>Information and communication technology managers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Computer manager</td>
</tr>
<tr>
<td></td>
<td>Computer operations manager</td>
</tr>
<tr>
<td></td>
<td>Data processing manager</td>
</tr>
<tr>
<td></td>
<td>IT manager</td>
</tr>
<tr>
<td></td>
<td>Systems manager</td>
</tr>
<tr>
<td></td>
<td>Telecom manager</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2131</th>
<th>IT strategy and planning professionals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Computer consultant</td>
</tr>
<tr>
<td></td>
<td>Software consultant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2132</th>
<th>Software professionals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Analyst-programmer</td>
</tr>
<tr>
<td></td>
<td>Computer programmer</td>
</tr>
<tr>
<td></td>
<td>Software engineer</td>
</tr>
<tr>
<td></td>
<td>Systems analyst</td>
</tr>
<tr>
<td></td>
<td>Systems designer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3131</th>
<th>IT operations technicians</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Computer operator</td>
</tr>
<tr>
<td></td>
<td>Database manager</td>
</tr>
<tr>
<td></td>
<td>IT technician</td>
</tr>
<tr>
<td></td>
<td>Network technician</td>
</tr>
<tr>
<td></td>
<td>Systems administrator</td>
</tr>
<tr>
<td></td>
<td>Webmaster</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3132</th>
<th>IT user support technicians</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Help desk operator</td>
</tr>
<tr>
<td></td>
<td>Helpline operator (computing)</td>
</tr>
<tr>
<td></td>
<td>IT helpline support officer</td>
</tr>
<tr>
<td></td>
<td>Support technician (computing)</td>
</tr>
<tr>
<td></td>
<td>Systems support officer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5245</th>
<th>Computer engineers, installation and maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Computer engineer</td>
</tr>
<tr>
<td></td>
<td>Computer maintenance engineer</td>
</tr>
<tr>
<td></td>
<td>Computer service engineer</td>
</tr>
<tr>
<td></td>
<td>Computer service technician</td>
</tr>
</tbody>
</table>

References

### Table 8: Job Profiles and Titles Included in Career Space.

<table>
<thead>
<tr>
<th>ID</th>
<th>Job title</th>
<th>Job Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Application Programmer</td>
<td>Software &amp; Applications Development</td>
</tr>
<tr>
<td>2</td>
<td>Software (SW) Engineer</td>
<td>Software &amp; Applications Development</td>
</tr>
<tr>
<td>3</td>
<td>System Developer</td>
<td>Software &amp; Applications Development</td>
</tr>
<tr>
<td>4</td>
<td>Technical System designer</td>
<td>Software &amp; Applications Development</td>
</tr>
<tr>
<td>5</td>
<td>SW Architect</td>
<td>Software &amp; Applications Development</td>
</tr>
<tr>
<td>6</td>
<td>Maintenance &amp; Support Specialist</td>
<td>Software &amp; Applications Development</td>
</tr>
<tr>
<td>7</td>
<td>Integration Technician</td>
<td>Software &amp; Applications Development</td>
</tr>
<tr>
<td>8</td>
<td>Software Programmer</td>
<td>Software Architecture and Design</td>
</tr>
<tr>
<td>9</td>
<td>Systems Developer</td>
<td>Software Architecture and Design</td>
</tr>
<tr>
<td>10</td>
<td>Systems Architect</td>
<td>Software Architecture and Design</td>
</tr>
<tr>
<td>11</td>
<td>Systems Architecture &amp; Design Scientist</td>
<td>Software Architecture and Design</td>
</tr>
<tr>
<td>12</td>
<td>Systems Integrator</td>
<td>Software Architecture and Design</td>
</tr>
<tr>
<td>13</td>
<td>Network Designer</td>
<td>Software Architecture and Design</td>
</tr>
<tr>
<td>14</td>
<td>Computer Scientist</td>
<td>Software Architecture and Design</td>
</tr>
<tr>
<td>15</td>
<td>Systems Integrator</td>
<td>Integration &amp; Test/Implementation &amp; Test Engineering</td>
</tr>
<tr>
<td>16</td>
<td>System Implementation Engineer</td>
<td>Integration &amp; Test/Implementation &amp; Test Engineering</td>
</tr>
<tr>
<td>17</td>
<td>Integration System Engineer</td>
<td>Integration &amp; Test/Implementation &amp; Test Engineering</td>
</tr>
<tr>
<td>18</td>
<td>Implementation Engineer</td>
<td>Integration &amp; Test/Implementation &amp; Test Engineering</td>
</tr>
<tr>
<td>19</td>
<td>Implementation and Test Specialist</td>
<td>Integration &amp; Test/Implementation &amp; Test Engineering</td>
</tr>
<tr>
<td>20</td>
<td>Integration and Test Specialist</td>
<td>Integration &amp; Test/Implementation &amp; Test Engineering</td>
</tr>
</tbody>
</table>

### References


### Table 9: EUCIP Professional Certifications.

<table>
<thead>
<tr>
<th>Title</th>
<th>Knowledge Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS Manager</td>
<td>PLAN</td>
</tr>
<tr>
<td>IS Quality Auditor</td>
<td>PLAN</td>
</tr>
<tr>
<td>Enterprise Solutions Consultant</td>
<td>PLAN</td>
</tr>
<tr>
<td>Business Analyst</td>
<td>PLAN</td>
</tr>
<tr>
<td>Logistics &amp; Automation Consultant</td>
<td>PLAN</td>
</tr>
<tr>
<td>Sales and Application Consultant</td>
<td>PLAN</td>
</tr>
<tr>
<td>Client Services Manager</td>
<td>PLAN</td>
</tr>
<tr>
<td>IS Project Manager</td>
<td>PLAN</td>
</tr>
<tr>
<td>IT Systems Architect</td>
<td>BUILD</td>
</tr>
<tr>
<td>Information Systems Analyst</td>
<td>BUILD / PLAN</td>
</tr>
<tr>
<td>Web &amp; Multimedia Master</td>
<td>BUILD</td>
</tr>
<tr>
<td>Systems Integration &amp; Testing Engineer</td>
<td>BUILD</td>
</tr>
<tr>
<td>Software Developer</td>
<td>BUILD</td>
</tr>
<tr>
<td>Database Manager</td>
<td>BUILD</td>
</tr>
<tr>
<td>X-Systems Technician</td>
<td>OPERATE</td>
</tr>
<tr>
<td>Telecommunications Engineer</td>
<td>OPERATE</td>
</tr>
<tr>
<td>Network Architect</td>
<td>OPERATE</td>
</tr>
<tr>
<td>Security Adviser</td>
<td>OPERATE</td>
</tr>
<tr>
<td>Network Manager</td>
<td>OPERATE</td>
</tr>
<tr>
<td>Configuration Manager</td>
<td>OPERATE</td>
</tr>
<tr>
<td>Help Desk Engineer</td>
<td>OPERATE</td>
</tr>
<tr>
<td>IT Trainer</td>
<td>OPERATE</td>
</tr>
</tbody>
</table>

Table 9: EUCIP Professional Certifications.
Software Patents in Europe after European Parliament’s Rejection

Alberto Barrionuevo-García

2006, Alberto Barrionuevo-García, under "Creative Commons Attribution-NonCommercial-NoDerivs 2.5 Spain License, Spain", <http://creativecommons.org/licenses/by-nc-nd/2.5/es/>

The success of software science and economics has been based on four legal pillars: copyright, trade secret, trademarks and, linked to copyright, end user licence agreements (EULAs). In the USA, since the early 80s, a controversial fifth legal regime, one which has never proved its worth in terms of benefit to society and software innovation, has been gradually emerging from case law: the software patent. However, not until the dot-com boom did software patents play a significant role in software economics. In Europe, where legislation is different from the USA’s, patents in general are regulated by the EPC, or European Patent Convention, EPC (Munich, 1973). This convention explicitly states that computer programs are non-patentable. However, the European Patent Office (EPO) has been prone to making fanciful interpretations in favour of the mass patentability of software, and indeed this remains their practice today. The few lawsuits of any importance that go to trial in the European Union are mostly found in favour of a more restrictive interpretation of software patentability, closer to the European Parliament’s contrary standpoint vis-à-vis the recently rejected software patent directive. Despite Spain having one of the most restrictive laws in Europe with regard to software patentability, there is still a certain degree of legal controversy created by the EPO’s rather sleight-of-hand interpretation of the EPC.


1 History and Legal Situation

1.1 History of The Intellectual Protection of Software

In the early days of Information Technologies (IT) and computing there was no economic separation between software and hardware. Software was a free add-on, without any economic value, provided with the hardware which was where the real commercial value lay. It could be said that in those days all software was public domain and open-source (it made no sense to make a secret of something which had no value).

During the sixties it became apparent that this situation was no longer tenable because software was providing an increasingly important amount of added value to the end solution, and the cost of creating software was rising all the time.

Given this scenario, industry and legislators met towards the end of the decade to decide on how to protect this...
new asset, software. There were three options: to apply copyright law, to use patents, or to develop a sui generis protection for software. The latter two options were ruled out and it was decided to use copyright. The main economic reason for patent protection being discarded was the fear that the dominant multinational at that time, IBM, would use its patents to monopolize the entire software market.

Thus, up until now copyright has been the mechanism by which all software is protected. Both the Berne Convention (1886-1979)\(^1\) and TRIPS (Trade-Related Aspects of Intellectual Property Rights, 1994)\(^2\), as well as the current European Directive (1991)\(^3\) and Spanish law (1996)\(^4\), maintain that, since they are intellectual creations, computer programs should be protected as if they were literary works.

With the commercial proliferation of closed-source software, copyright was joined by three further legal mechanisms: trade secret, trademarks and, as an offshoot of copyright, user licence agreements (EULAs).

The present situation is that proprietary software is regulated by the four abovementioned mechanisms, and libre (free) software is only protected by copyright, for example by copyleft or related EULAs (GPL, or General Public License; BSD, or Berkeley Software Distribution, etc.).

1.2 History of The Software Patent

1.2.1 United States of America

In the USA there is a legal concept that does not exist either in the same way or with the same importance in continental Europe, although it does in the United Kingdom: case law. In the USA, the interpretation of a law, or a ruling if no law exists, given by the first judge to rule on a new legal issue creates case law, and his or her ruling will have great weight as a precedent for subsequent cases dealing with the same issue.

In continental Europe, however, in similar circumstances it is common for various judges to make independent rulings without reference to previous cases.

Furthermore, with regard to patents, the European requirement of having to deal with technical aspects does not exist in the USA, with the result that it is normal to grant a patent for any new invention that may be useful.

At first the US Patent Office was not in favour of granting software patents, arguing that they lacked the specialist staff required to examine them. Over time, that decision gradually changed, aided by pressure from some of the Patent Office’s most important customers (for a long time IBM was its best customer in terms of the number of patents granted).

But whatever the subject matter, patents are not much use if their holders cannot enforce them and sue those who infringe the 20 year monopolies that they confer. Thus, given the lack of any specific legislation governing software patents, US courts created a body of case law, mainly in the 80s, in favour of software patents. In this respect there have been two key rulings: when the Supreme Court in 1981 decided to ratify the validity of a patent for software with a practical application, and in 1998 when it decided that "everything under the sun that is made by man (sic)" is patentable.

However, it was not until the dot-com boom that software patents began to play a major role in the economy of software, not if we judge the importance of that role by a) the amount of damages awarded for infringement by US courts, running into the hundreds of million dollars; b) the number of applications for software patents; and c) the number of infringement trials, lawsuits, and notices being filed. In a recent study, professor Brian Kahin of the University of Michigan estimated that 100 cases a year go to trial in the USA from 2,500 lawsuits, and he extrapolated that if the "rule of 25" were to be applied, that would mean 5 million notice of infringement letters and licence demands that did not lead to litigation.

The company holding the most software patents in both the USA and Europe is currently IBM, while the company with the most patent applications is Microsoft.

1.2.2 European Union

In continental Europe, legislation and the concept of case law is different from how it is in the USA. Generally speaking, throughout Europe patents are regulated by the 1973 European Patent Convention (EPC)\(^5\). Its article 52.2 states explicitly that computer programs are not susceptible to patent protection, although point 52.3 has given rise to highly fanciful interpretations by the European Patents Office (EPO).

The legislator’s intention in point 52.3 was to say that, although software itself is not patentable, inventions that incorporate software could be patentable, provided that the software was not the subject matter of the patent (software as such or software per se). Unfortunately, the manner of expressing this idea by means of a simple “as such” was not sufficiently categorical, and this, together with (and mainly due to) the unsatisfactory manner in which the EPO was set up, has given rise to most of the problems regarding software patents in the European Union today.

The EPO was set up as a completely autonomous organization, under the

---

sole legal auspices of the EPC and with no other controlling legal authority. At first it interpreted those two abovementioned articles according to the spirit of the legislator, as can be seen in its own 1978 guidelines in which it describes a computer program in the following way: "A computer program make take various forms, e.g., an algorithm, a flow-chart, a series of coded instructions which can be recorded on a tape or other machine-readable record medium, and can be regarded as a particular case of either a mathematical method or a presentation of information".

This description of a computer program totally ruled out (intangible) software as a patentable subject matter, while it opened the door to patenting "traditional" (tangible) inventions that included software, provided that the software "as such" was not the subject matter of the patent.

However, by mutual agreement with the other two members of the "The Trilateral Cooperation" (the US and Japanese patent offices), the EPO has been gradually changing the way it interprets its own founding charter and has been falling in line with the demands of its main customers: major tech and software multinationals from, in order of importance, the USA, Japan, and Europe.

The EPO’s current doctrine is to consider that "computer programs" "as such" only refer to source code and target code when they are not being run on a computer, and that any idea that can be implemented on a computer by means of software (computer programs), whether it be software, algorithms, business methods, functionalities of the software or mere functional processes, can be patented, because it displays the technical aspect of the use of a computer (notice that this technical aspect is equivalent to that found in human thoughts or in writing on paper). In practice, this doctrine nullifies and makes a mockery of article 52.2 of the EPC, which is why it is bypassed so often.

This is reflected in the 2001 guidelines which state that "a further technical effect which goes beyond the normal physical interactions between the program and the computer" is patentable, but does not go on to define either what it is "further" to, what is meant by "technical", how far "beyond" it has to go, or what is deemed to be "normal". The 2001 guidelines are also notorious for introducing the well-worn euphemism "computer implemented invention", imported from the 2000 Trilateral Conference, to allow any idea implemented by computerized means to be patented, regardless of whether or not it is of a technical nature (such as business methods). Clearly, the only thing that can be implemented in a computer is software (the programs) which, in short, means that this term is merely a euphemistic way of referring to software without actually having to call it by its name, and therefore preventing it from falling into the category of exempt, non-patentable, subject matter.

However, the problem for the EPO is that the more than 30,000, <http://gauss.ffii.org>, software patents that it has granted are hard to enforce under the national legislations of the member countries of the EPO where, generally speaking, the laws are closer to

Figure 1: Patent on Internet Shop.
the EPC’s original intention than to EPO’s present doctrine. Generally, and not uncontroversially, the rulings of the few cases to go to court on this issue in the European Union have mostly been based on a more restrictive interpretation of software patentability, one closer to the contrary standpoint taken by the European Parliament opposing software patents in the first reading of the software patents directive, which was ultimately almost unanimously rejected.

This situation gave rise to two serious, albeit failed, attempts by the EPO to change current legislation: the first in 2000 via its founding charter, the EPC; and the second in 2002 with the rejected EU software patents directive that was intended to "harmonize" all the national laws with the EPO doctrine and US practice (in practical terms almost identical), and that caused such social unease.

This lack of coordination between the EPO and national legislations has left European software patents with no legal recognition, such as the example shown in Figure 1 (see [1] in Annex), or patents on word processor in Figure 2 (see [2] in Annex).

The chart in Figure 3 shows the country-by-country distribution of software patents granted by the EPO (see Figure 2).

1.2.3 Spain
Under Spanish patent law, one of the most restrictive laws regarding software in Europe, all patent claims for inventions that "comprise" (contain) computer programs (Law 11/1986) are invalid provided that such invalidity does not contravene EPC (Royal Decree 2424/1986).

Thus, in their interpretations, Spanish courts are ultimately not to contradict the EPC text, but neither do they follow the EPO’s sleight-of-hand interpretive doctrine.

Edo el arcu eu dolor

Spanish patent law also says that patents should "raise the level of competitiveness of our industry" and "effectively protect the results of our research". Bearing in mind that Spain accounts for no more than 0.20% of all software patents granted by the EPO (see Figure 3), while USA and Japan have more than 70%, it should come as no surprise from a political and economic viewpoint that the Spanish representative voted against the text of the proposed new directive in the controversial and manipulated EU Council of May 17, 2004, under Irish Presidency.

1.3 Current Situation
1.3.1 United States of America
Software patents and, to a lesser extent, patents in general, are being increasingly questioned in the USA insofar as they act as a barrier to competitiveness and innovation, while benefiting a handful of dominant companies which are themselves not encouraged to innovate or compete. Significantly, a report issued in 2003 by the US Federal Trade Commission questioned the appropriateness of patent protection in its present form.

In any event, on the subject of software patents, US politicians are more
concerned with what is happening on the other side of the pond than events in their own backyard.

### 1.3.2 European Union

After the overwhelming rejection of the software patent directive in July 6 2005 by the European Union’s sole representative body, the EU Parliament, pro-software patent groups are once again pressuring to legalize them by means of the recently "revived" Community Patent Directive. Up until now, this directive has been a major source of conflict and the various countries involved have never achieved the consensus required for the directive to be passed.

The main sticking point has been what authorized languages are to be used to file a patent application, and more specifically, whether translations of patent specifications are to be binding or merely informative.

The way to legalize software patents under this directive would be based on the following premises included in the current draft directive issued by commissioner McCreevy, responsible for the European Commission Internal Market and Services DG (the same official who presented and defended the software patent directive):

1. Community patents would be granted and managed by the European Patent Office.
2. The Community Patent Directive would codify EPO’s current administrative practice as law. As far as software patents are concerned, this practice is virtually a copy of what the European Parliament has already rejected and amended in two successive readings. The codification of this law could, in itself, be enough to legalize software patents in the European Union.
3. A European Court of Appeal would be set up solely to deal with patents. This court would be composed, managed, and controlled or highly influenced by the EPO and would be empowered to apply case law to give a final ruling on the legality of software patents in the EU.

It should be remembered that the EPO does not belong to the European Union, so the fact of granting such a degree of autonomy to this external office staffed by functionaries could even be construed as an infringement of the constitutional treaties of the European Union itself.

Disapproving voices can also be heard in traditional judicial circles, where they object to the constitution of a special court that would be under EPO control rather than the independent control of a legitimate judicial body.

If the proposed directive comes into force in its present form, not only would software patents be legalized, but the EPO—an external, autonomous, independent administrative authority with no oversight—would be granted executive, legislative, and judicial power over patent matters, and consequently, control over technological innovation and competitiveness in the European Union. This would probably put the EPO up on a par with all the ministries of industry in terms of control over Europe’s productive fabric and economy.

It remains to be decided how this new directive would be legislated: whether by codecision between the European Council and the European Parliament, or by the hitherto failed unanimity voting procedure among members of the Council of the European Union.

Depending on the path the Commission eventually takes, the legislative procedure will either be very fast (a matter of a few months) or as slow as the now defunct software patent directive (several years).

In any event, it has to be said that the Community Patent Directive is, in itself, intrinsically neither bad nor good. The problem is that, depending on how it is enacted, it could mean either the definitive legalization of software patents in the European Union (and worldwide), or else their abolition and the channelling of the EPO, or any office that may replace it, towards democratic standards under the control of the legitimate representative authorities of the EU.

### 2 Economy

#### 2.1 Software SMEs

The lion’s share of IT business and innovation in Europe is in the hands of SMEs (Small & Medium size Enterprises), while the large multinational-
tors” in the market that are infringing those patents or by selling them to the highest bidder. Patent trolls are especially aggressive, demanding large sums from hardware and software companies. They tend to make a great deal of money from small companies for whom consulting a patent lawyer would end up being more expensive than paying for the licence.

The problem for society is that the companies that end up paying out for these licences (or extortion payments if you like) are precisely the companies that are truly productive and innovative: customer oriented companies that are useful to society.

Any benefit that these companies may obtain from taking out a software patent will be totally wiped out by the cost of making licence (i.e. extortion) payments for all the patents that the software they are producing will almost certainly infringe, even without them knowing that such patents exist.

Meanwhile, the possible benefit to be obtained by the dissemination of innovations that the patent system supposedly entails (to patent originally meant “to make public”) is more than debatable. How many IT professionals consult the US software patent database (the largest in the world with over 300,000 entries) while designing their software solutions? None, I would hazard to guess.

Software patents do not appear to bring much in the way of innovation either to SMES or to society, but they do open the door to a flourishing extortion racket against true software producers and innovators.

2.2 Large Software Multinationals

Large software and technology multinationals (nearly all of which are users of software) have a resource that SMEs do not have: the possibility of entering into large scale patent swap deals. This possibility is clearly open only to a select club of major companies: it is unlikely that IBM would be very interested in swapping its thousands of software patents for a handful held by an SME.

This group of companies can be divided into two main types: those that have a de facto monopoly in a more or less broad market niche and those that do not.

For those companies without very large niche market shares, what they gain with their patents and what they lose through patent infringements could actually balance out, although they always live with the uncertainty that their main asset, their software, may infringe patents that are impossible to work around, for which they will be obliged to pay tens or hundreds of million dollars that they have not budgeted for: e.g. Kodak vs. Sun, e-Plus vs. Ariba, NTP vs. RIM (Blackberry), etc.

For those companies with monopolies or very large niche market shares (Microsoft, Oracle, SAP, Cisco, Adobe, etc.), software patents can be a way for them to maintain their monopoly by preventing their competitors from gaining access to software resources that in some cases may be indispensable. An example of this is Microsoft publicly threatening Linux for infringing its software patents, or their refusal to release open-source code as part of its licensing agreements. Weighed against the benefit of maintaining or even increasing an already near monopolistic market share, paying several hundred million every now and again is a perfectly acceptable risk: e.g. Eolas vs. Microsoft, e-Plus vs. SAP, etc.

Also, the application and litigation costs of software patents are not an excessive burden for any of the major companies, which gives them a tremendous advantage over their SME competitors in the software market.

2.3 Software Solutions of Non-IT Companies

This type of software providers would also be obliged to either pay licence fees or not infringe present and future software patents.

These companies are faced with a serious problem, because their developments tend to be known by a large number of developers who are either not on the company payroll or some day will cease to be on it (a lot can happen in 20 years); people who one day may be working in a consulting firm and the next in some other company.

This makes it fairly likely that the patent owners will get to know of the existence of these developments and may end up asking for large sums of money for the infringed patents. Ex-employees may even try to extort money from them by threatening to reveal patent infringements.

This risk is even higher for companies that provide mass market services, such as telephony or ASP/ISP (Application Service Provider/Internet Service Provider), because compensatory payments will be in proportion to the number of end users.

2.4 Libre Software, Shareware, Freeware and Other Models of Free Software Distribution

2.4.1 Pay per Copy

This type of software producers or free Internet software distribution models are at a major competitive disadvantage to other distribution models. These software producers have no way of knowing how many copies of a given software package there are on the market. It is therefore impossible to quantify the damages or the number of licences per copy that they would have to pay in the not unusual event of their infringing a patent.

2.4.2 Interoperability and Standards

The existence of patents also involves the licensing of the standards used in the inevitable pursuit of software interoperability. Practically all software has to interoperate to a greater or lesser extent.

Owners of software patents can be required by law or, say, by a standardization consortium, to adopt one of three policies with regard to licensing software in order to permit interoperability: 1) without any restrictions or obligations in terms of licensing, 2) RAND (“Reasonable and non-discriminatory”) licences and, 3) libre, open, and free licences.

The first type allows the holder of the patent on the interface to license it or not, and at the price and under the conditions they see fit for each customer. This can be seen as a way to selectively exclude or severely penalize the patent holder’s competitors. This form of licensing is out of the question for any object with the remotest chance of being called a standard.

The second type, RAND, to a cer-
taint extent commits the patent holder to license their software at a price and under conditions which are public, standard and, in theory, fair. However, RAND licensing, contrary to what its name (another euphemism) implies, is in fact discriminatory, since it can easily prevent libre distribution models, or models whereby the code is made public thus revealing interoperability information, from interoperating with the patented interface (e.g. European Commission vs. Microsoft). Licences of this type are used for some international standards, not without the general disapproval of the communities discriminated against: generally libre software, open-source and, sometimes, the other free software distribution models.

The W3C (World Wide Web Consortium), responsible for standardizing the Web, had to reverse its decision to use this licensing model in its standards so as not to discriminate against libre software or free software practitioners.

It is significant that during the legislative procedure of the rejected software patent directive there was a wide-ranging debate, especially among the defenders of software patentability, about allowing interoperability to be exempt from infringement.

The third option is to take the exemption of interoperability to the point at which patent monopoly would be abolished in the case of standards. This is mandatory in the case of some standardization consortiums such as the aforementioned W3C, OASIS (Organization for the Advancement of Structured Information Standards) with their OpenDocument, or even by law. If software patents are to exist legally in the market, this model is the only one that allows free and open standards; that is, the only one that is really non-discriminatory against any kind of software or group.

2.4.3 Open-source and Not-for-profit Development

Open-source software, whether libre or proprietary, runs a greater risk of being sued for patent infringement. It is much easier for patent holders to check or find developments that are in breach of their patents as they have all the source code at their disposal.

The other intrinsic problem that libre software has with patents is that a large part of its development is carried out by volunteer programmers that either cannot or are not willing to pay thousands of euros to take out patents. Ultimately, their way of working is based on sharing and collaboration and is therefore, by nature, diametrically opposed to filing patents to prevent others from using their software. This is why libre software practitioners are at a competitive disadvantage, because although they cannot patent their innovations they still have to comply with everyone else’s patents.

Some large multinationals, however, have released part of their patent portfolios to be freely used by either all the libre software community or, occasionally, by a very limited part of it (IBM, Sun, CA, Nokia, etc.)

Late in 2005, some of the companies in favour of software patents but interested in libre software formed a company to manage patents licensed to libre software practitioners, mainly Linux.

However we do not expect this to be a solution, nor do we think it will do much to minimize the current threat in markets where software patents are legal, since any project could still be taken to court for infringing any of the other patents that it cannot obtain licences for.

Meanwhile, it is not at all unusual for libre software projects to receive letters from patent holders demanding that the call a halt to their developments. The threats received by the Videolan (VLC) and mPlayer multimedia player projects, both based in Europe, are common knowledge.

But basically, in the markets in which software patents are legal, those who have most to fear economically from these lawsuits are large companies using open-source software.

3 The Essence of Software

The fundamental difference between software and patentable subject matters is their intellectual or intangible nature. Patents have always been applied to physical implementations of ideas as described in patent applications. Thus knowledge of an invention is made available in exchange for a temporary monopoly. In these cases what is patented is not the description of the invention, but the specific combination of physical components making up that device as included in the description and in the patent claim.

In the case of software this combination of devices is always the same: a computer or various computers.

The underlying problem of patenting software is that we are dealing with something that is intangible and intellectual, the implementation of which will coincide with the description if that description is sufficiently precise and accurate. What better and more precise way is there of describing a software idea than by writing its source code?

Thus, when we patent the description, whether it be imprecise or highly detailed, what we are patenting and monopolizing is the description of the idea. In other words, the patent is protecting the idea itself and not what it is supposed to be protecting, which is the implementation of an idea.

This contradiction leads to incongruities such as the fact that, in practice, software patents can have the effect of usurping ownership or, at the very least, are in conflict with copyright protection of software. Who does a software development rightly belong to, its author or the holders of the software patents that it infringes?

There are also ridiculous situations, such as the fact that, in order to develop software safely to avoid being sued for infringements, you need to search the entire database of software patents and patent applications to ensure that you are not committing an infringement.

Practically any remotely useful computer program nowadays will infringe some software patent, and usually a large number of them. More serious still is the fact that you can never be sure of whether or not you are infringing many of the patents granted due to the deliberately cryptic and arcane language used in software patent applications. Plus the fact that there is sometimes no way of getting around patents if you want to develop something.

Another incongruity which is even more self-evident is the 20 year life of
evolution. Will we have to wait 20 years before we can properly promote and popularize IPv6 to be sure that our networks will not be on the receiving end of a massive software patent lawsuit?

Bizarre way to encourage innovation!

4 Conclusion

Software patents, apart from having been rejected on economic, democratic, and social grounds in the European Union, currently occupy a legal no-mans-land at best.

They are mainly a sub-product arising out of the bodged constitution of the European Patent Office, a supranational authority which, in practical terms, is outside any democratic control and which wields (at least partially) executive, legislative, and judicial control over matters of patents.

The main problem is that, in practical terms, software patents leave the market of software creation and innovation in the hands of a few large multinationals, most of which are non-European, without there being any benefit to innovation in general.

In regions like Europe, where the software creation market is almost wholly dominated by SMEs and which is also largely underdeveloped compared to the USA, this poses a serious problem of competitiveness and creation and may even affect the maintenance of the local industrial fabric. Thus EPO’s moves in favour of harmonizing patent legislation with the USA (and Japan) with regard to patenting software and abstract or intellectual subject matters would favour EPO’s best “customers” but would seriously harm the interests of the rest of the society that it was supposed to be serving.

If we want to preserve the health of innovation and innovators in Europe, it is necessary to undertake a major reform of the EPO, so that it remains under the control of the democratic political authorities of the European Union and to ensure that economic uncertainty is not allowed to return to the European software market.

Finally, the greatest danger: if software patents were to be legalized, the first patents on intellectual subject matters would open up a dangerous legislative breach.

What areas of human knowledge would be the next to be monopolized? The sciences, the arts, human genetics, law, religion...?

Translation by Steve Turpin

Bibliography and Documentary References

There is scant documentation available on software patents in book form, but there is a great deal of information to be found on the Internet, such as the following pieces:


