Pre-Proceedings

IEEE International Workshop on
Software Technology and Engineering Practice

STEP 2005

September 24-25, 2005
Budapest, Hungary

Sponsored by

IEEE Computer Society
Research Centre on Software Technology Dept. of Engineering, University of Sannio
Department of Electrical and Computer Engineering, Queen's University
University of Waterloo
Reengineering Forum
Software Engineering Institute

Pre-Proceedings

IEEE International Workshop on Software Technology and Engineering Practice (STEP 2005)

September 24-25, 2005
Budapest, Hungary

Edited by
Ying Zou
Massimiliano Di Penta

Sponsored by

IEEE Computer Society

Research Centre on Software Technology
Dept. of Engineering
University of Sannio, Italy

Department of Electrical and Computer Engineering
Queen's University, Canada

University of Waterloo, Canada

Reengineering Forum

Software Engineering Institute
Carnegie Mellon University, USA

Contents

IEEE International Workshop on
Software Technology and Engineering Practice

STEP 2005

Keynote

1.1 Workshop on Software Process Improvement, Quality Assurance and Measurements

2.1 Workshop on Evolution of Software Systems in a Business Context

Tools and Architectures for Evolution:

Reverse Engineering of Business Rules:

2.2 Workshop on Empirical Studies in Reverse Engineering

2.3 Workshop on Architecture Recovery towards Reuse

2.4 Workshop on Design Issues for Software Analysis and Maintenance Tools

3.1 Workshop on Net-Centric Computing (NCC 2005): Middleware -- The Next Generation

4.1 Workshop on Design Patterns Theory and Practice

4.2 Workshop on System Integration and Interoperability

Message From the General Chair and Program Chairs

STEP 2005

It is our pleasure to welcome you in the magnificent atmosphere of Budapest for STEP 2005, the 13th IEEE International Workshop on Software Technology and Engineering Practice.

STEP is a workshop-style event, intended to offer a unique opportunity to blend the perspectives of software practitioners and university researchers in the use of advanced software engineering practices. Each workshop reviews and extends a roadmap for software technologies with contributions on new development, practical experience, technology evaluation and gap analyses.

This year STEP provides the opportunity to participate to eight workshops, dealing with four different themes: process and measurements, system analysis and evolution, network computing and emerging technologies.

Overall, the workshops received a total of 54 position papers from 20 different countries, ranging over all the five continents: this will make STEP 2005 a truly worldwide, exciting event. We believe that the presentations and the discussion will contribute to identify challenges develop ideas and provide approaches. Also, it is our hope that the upcoming discussion will stimulate authors and workshop participants to continue and improve their work, and to submit high-quality, full technical papers for the STEP 2005 post-proceedings.

We would like to thank the authors for submitting position papers and coming to STEP, and the workshop organizers for the great work they have done in advertising their workshops, soliciting papers and setting up a great program. Special thanks deserve all the people of the organizing committee and of the steering committee for making this event possible. Finally, we would like to acknowledge our key sponsors: the IEEE Computer Society, the Research Centre on Software Technology (RCOST), the Department of Electrical and Computer Engineering of Queen's University, the University of Waterloo, the Reengineering Forum, and the Software Engineering Institute.

We wish you enjoy the program and have a pleasant stay in Budapest!

Kostas Kontogiannis, Ying Zou and Massimiliano Di Penta
STEP 2005 General Chair and Program Chairs

Conference Committee

STEP 2005

General Chair

Kostas Kontogiannis
Department of Electrical & Computer Engineering, University of Waterloo, Canada

Program Chairs

Ying Zou
Department of Electrical & Computer Engineering, Queen's University, Canada

Massimiliano Di Penta
RCOST, Department of Engineering, University of Sannio, Italy

Publicity Chair

Liam O'Brien
Software Engineering Institute, USA

Local Arrangements **(assisted by)**

Andrea Tosokyne
SoRing Ltd, Hungary

Finance Chair

Dennis Smith
Software Engineering Institute, USA

Workshops List

STEP 2005

1.1 Workshop on Software Process Improvement, Quality Assurance and Measurement (Sept. 24, 2005, Morning)**,** organized by:

Jos J.M. Trienekens, TU Eindhoven, The Netherlands
Rob Kusters, TU Eindhoven, The Netherlands
KEMA Arnhem, The Netherlands
Katalin Balla, TU Budapest, Software Quality Institute, Hungary

2.1 Workshop on Evolution of Software Systems in a Business Context (Sept. 25, 2005, full day)**,** organized by:

Keith Bennett, University of Durham, United Kingdom
Damiano Distante, University of Sannio – RCOST, Italy
Maria Tortorella, University of Sannio – RCOST, Italy

2.2 Workshop on Empirical Studies in Reverse Engineering (Sept. 24, 2005, Afternoon)**,** organized by:

Paolo Tonella, ITC-irst, Centro per la Ricerca Scientifica e Tecnologica, Italy

2.3 Workshop on Architecture Recovery Towards Reuse (Sept. 24, 2005, Morning), organized by:

Chris Exton, University of Limerick, Ireland
Jim Buckley, University of Limerick, Ireland
Liam O'Brien, Software Engineering Institute, USA

2.4 Workshop on Design Issues for Software Analysis and Maintenance Tools (Sept. 24, 2005, Afternoon)**,** organized by:

Dean Jin, University of Manitoba, Canada

3.1 Workshop on Net-Centric Computing (NCC 2005): Middleware -- The Next Generation (Sept. 25, 2005, Morning)**,** organized by:

Scott Tilley, Florida Institute of Technology, USA
Kostas Kontogiannis, University of Waterloo, Canada
Sihong Huang, Florida Atlantic University, USA

4.1 Workshop on Design Patterns Theory and Practice (Sept. 25, 2005, Morning)**,** organized by:

Giuliano Antoniol, Ecole Polytechnique de Montreal and University of Sannio
Yann-Gaël Guéhéneuc, University of Montreal, Canada

4.2 Workshop on System Integration and Interoperability (Sept. 25, 2005, Afternoon), organized by:

Dennis Smith, Software Engineering Institute, USA
Liam O'Brien, Software Engineering Institute, USA

Keynote

Changing Paradigm of Software Development

Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202, U.S.A.
rajlich@wayne.edu

Software evolution is the part of the software lifecycle in which substantial new functionality is introduced into software. It plays a prominent role in the software lifecycle, and the new development techniques of incremental, agile, and extreme programming further emphasize its role. The requirements volatility provides the rationale for the increasing role of software evolution and these new development techniques.

Software refactoring and incremental change comprise the core of software evolution. While refactoring repairs the structure of the evolving software, incremental change changes the functionality. The mini-cycle of incremental change contains change request, design, implementation, validation, and release. While incremental change has been practiced since the beginning of software, many interesting challenges are still open.

***

Václav T. Rajlich is a full professor and former chair in the Department of Computer Science at Wayne State University. He published extensively in the areas of software engineering, evolution, and comprehension. He a current program co-chair and former general chair and steering committee chair of IEEE International Conference on Software Maintenance. He is also the founder and former general chair of IEEE International Conference on Program Comprehension. He received a PhD in mathematics from Case Western Reserve University. Contact him at rajlich@wayne.edu.

Workshop 1.1:
Workshop on Software Process Improvement, Quality Assurance and Measurements

Software process improvement, quality assurance and measurement

Dr. Katalin Balla

In the intense international competition software companies are more and more forced to think about proving their capability of delivering good products. One way of having such a proof is to produce an official certificate about usage of a certain standard or model. However, introducing an approach based on a standard or model, and institutionalising it, so that the organisation is able to pass an audit, requires a lot of investment from software companies, both in terms of money and effort – which a company would not like to waste. Therefore, really business-driven software companies will be willing to do only *really efficient* software process improvement.

In our opinion, efficient improvement programs are always based on real needs of companies, will always start from understanding the actual situation. Choosing the right approach, model or standard for the improvement program would be the next step.

The difficult question is: which model to choose to best fit the company's needs in improving software quality? For answering this question, we have to understand shat software quality means in each particular situation.

Quality of software is a very complex subject, and, as such, it is extremely hard to define. If we wish to deal with software quality in its complexity, we have to think about the software products, the processes that produce the products and the resources that execute the processes. We have to define these objects, to choose the right quality attributes for them and verify their actual value by the means of objective metrics.

The approaches, standards and models used in software industry are extremely various in their approach used. The *process* of producing software is being still very emphasized (by models like CMM / SPICE / CMMI, by ISO 9001:2000, AQAP, ISO 12207, by project management methodologies and software development methodologies), but approaches concentrating on software *product* characteristics and metrics (like ISO 9126 family) are also being more and more accepted, together with models and approaches emphasizing the importance of the *human factor* (e.g. P-CMM, PSP, TSP). As no approach, model or standard covers all the aspects of software quality (although new versions of earlier models are definitely more broad in their scope, in the number of objects they are dealing with), companies will have to choose the right approach based on their business needs. Understanding the business needs in a right way is a rather complex job that claims solid professionalism both in the field of software development's nature and existing quality models and standards. Choosing a wrong approach could do considerable harm to a company, by misleading the efforts from the really important objectives.

One way of avoiding the trap of a badly chosen quality model or approach is to quit exclusively relying on *one* certain quality model - in favour of choosing among several approaches, *using more approaches in a synergic way*, according to the specific business needs of a certain organisation.

Such a combined use of more quality models or approaches would be based, on the one hand, on a solid theoretical knowledge and understanding of the popular quality models and their interconnections, and, on the other hand, on a huge experience in the way the companies in case execute their development processes and do business.

We consider that the theoretical framework for starting an efficient model-based software process improvement program as well as the concrete experience in the domain is worth to be presented to specialists being in connection with software quality issues.

After underpinning the above statements in a more detailed way, in the STEP 2005 Workshop we present a possible framework (QMIM – Quality through Managed Improvement and Measurement) helping to deal with software quality in its complexity. The framework has well defined elements, static and dynamic aspects, offering an approach to choose among different quality models.

In the second part of the presentation we show the way how the principles and elements of QMIM can be used in (Hungarian) software companies. The case study presented is based on the author's 12 years of experience in software process improvement. We briefly present some successful SPI programs that resulted in ISO certificates, and we show the "usual" quality- related trials "after - ISO", the elements of software quality the companies can deal with. We present concrete results of using QMIM ideas, as well as concrete elements of different quality models combined in order to define a company's "own" software quality.

In our experience, software process improvement results in Hungary are more and more being used in the SPI programs started to implement CMM, SPICE or CMMI in the companies. We show the results of formal and informal CMM / CMMI-based assessments in Hungary and the most important successes of such projects, together with the major difficulties Hungarian companies normally face. We analyse the cause of those difficulties.

Further position papers and discussions in the workshop will contribute to a more detailed picture of a possible way to do software quality improvement: using more quality models in a synergic way.

As SQI- Hungarian Software Quality Institute, together with the Technical University Budapest and Technical University of Eindhoven works on developing a "market-ready" QMIM model and methodology, the feedback obtained on the workshop will be used to develop a tool that is really helping specialists in choosing and using the software quality model that is best fitting their own needs.

Achieving CMMI 1 – How to develop a team that improves its own processes

Molnar Atilla

Case study

The definition of SEI CMMI Level 1 implies that organizations at this level are able to change, otherwise they cannot achieve CMMI 2. This means that there is a “hidden” level, at which the members of the team are in such a human condition that the organization cannot change.
Usually this happens after a long period of organizational changes, when no clear goals are set, the team has no management support and therefore the team is being isolated from the company as a whole. They feel they are left alone and will try to set their own goals (but this goals are not consolidated with the company goals therefore cannot be achieved *ab ovo*).
There is no formal structure of the team, no communication between the company and the team.
This situation is very bad for the members as human beings and lead to counter-selection.
Two possible solutions are possible.
First, the manager can use quick and dirty solutions with “military” tools. It can work but has very high risks and almost certainly the most of the team will leave the company. So at the end we will have a completely new team socialized with military style management.
The second solution is that we help the team improve their own personal and organizational “maturity” so that they slowly change in a positive way – this method is less risky but much slower. Our experience is that this way the members of the team will also change; but the new organization will be socialized to “self-improve”, so we call this method “organic”.
In order to do this, rather classical management tools are used:
1, Stabilizing the critical projects
At this step we should find the “key personnel”, and we should motivate them, so they will be able to produce heroic and impressive results within a short time (this is a requirement in CMMI Level 1) Also, this will sort of fix our positions at the customers
2, Creating strategy
It is very important to have a strategy, both short-term and long-term. We should keep in mind that at the beginning, members of the team will not belive in it – because they feel that it is “yet another nagging of them”. But the manager should be very committed with the strategy – otherwise a new chaotic situation can be achieved.
The strategy should contain the process development (i.e. CMMI) in the very early stage; otherwise at a later time it will be hard to add this element for various reasons.
3, Starting communication
The most important direction of “new communication” is the company. It takes a very long time for the company to trust us, so we should start it as early as possible.
The situation is the same with the customers.
And finally, the manager should improve dramatically the internal communication in the team.
Using this method, first the key persons will feel that things are changing the good way; they will start believing that the manager knows his job. They will motivate the others.

When the key persons are committed, the manager's job will change. He should listen to the wishes of this persons and answer their requests. So, the process development will be based on their (internalized) wills, which is much more efficient.
There will be a point when the wishes of the team are so many that they can be handled using some methodology – this is the third phase. At this period, the manager should point to the long term strategy that is set at the beginning and say "it's time to start with (i.e.) CMMI". And the team will happily do that without major organizational resistance.
The reason for this is that the organizational goals are conform with the members' own goals; the internal vision statement of the strategy (such as "we would like to earn a lot of money with professional work", "we want a liveable workplace" etc.) can be communicated frequently.
Finally, a formal process improvement project can be set up.

Matching management practices with IT analyst job motivation factors.

Fie W.T. De Wit[1]
Fred J. Heemstra[1]
Rob J. Kusters[1,2]
Jos J.M. Trienekens[2]

[1] : Open University, The Netherlands
[2] : Eindhoven University of Technology, The Netherlands

Introduction

Software development is a largely cognitive and therefore 'invisible' activity. Software professionals spent a significant part of their time working alone and thinking (Sullivan, 1988). It is also known that staff quality has a large impact on software productivity (Boehm et al, 2000). In such an environment staff motivation will play an important role. Improvements in sdtaff motivation are more than likely to improve product quality.

If management is to make use of this, their understanding of what motivates IT professionals should match the perceptions of this IT staff. This paper describes development and usage of an instrument to test if such a match in perceptions between IT staff and IT management exists.

This research was carried out at the internal IT department of a large retail organisation. This IT department had just been reorganised in such a way that a product line organisation was replace by a more functional organisation format where information analysts were now grouped in a single department. In the previous product based organisation their main motivational criterion had always been 'to get the job done'. Obviously, in a functional organisation, different management approaches might be required.

This prompted this project where a survey instrument was developed, aimed at both information analysts and their managers. Information analysts were asked what motivated them and managers were asked how they acted in order to motivate their staff. By comparing these results, an indication of how well this matched was obtained.

In this paper we will first describe the research approach taken. (XXXX)

Research approach

A first step was to identify from literature what motivation factors to include in this project. A significant body of literature discusses this issue of staff motivation. Already in 1959 Herzberg et. al. identified and tested a list of factors that either positively (satisfiers) or negatively (dissatisfiers) influence staff motivation. In 1976, Hackman and Oldham published their seminal job characteristics model, which builds on Herzberg's work and is still regarded as a major model in this area.

Given the heavy dependency of software engineering work on the availability of knowledgeable and motivated staff, in comes as no surprise that specific enquires into job motivation for IT professionals also took place. An application of Herzberg's work for software development professionals was reported by Fitz-enz in 1978. Cougar and Zawacki (1980) applied the Hackman and Oldham model, looked specifically at differences between clerical, professional and managerial workers in the software industry. The work of Couger and Zawacki was extended by Goldstein and Rockart (1984) and, almost simultaneously, by Ferratt and Short (1986). Since then, further research has looked in more detail at the motivation factors identified, but in essence these models still command respect.

We decided to take both the overview models as proposed by Goldstein and Rockart (1984) and by Ferratt and Short (1986) as a starting point for this research. Both models have been validated in practice, have shown their value over time and are sufficiently manageable to use in this type of research. Also, since each model adds to the original Couger and Zawacki model from a different perspective, they together provide a good overview of possible motivation factors.

We also decided to use a survey format for this project since this provided the possibility to present information analysts and managers with an explicit, written reference that would enable comparison of results between the two groups. Interviews using a form was considered as an alternative because it would offer to possibility to recognise and correct misunderstandings and to obtain additional comments to support the answers given. However this option was rejected because of the additional costs (time) this would entail for the organisation. The reduce possible adverse effects of this decision a carefully designed set of definitions was tested and provided.

The survey instrument

The main part of the instrument is formed by the survey instrument provided by Ferratt and Short (1986). We selected this instrument because its acceptance and its ease of use. Ease of use is an important criterion in the context of this research since its goal is to provide IT management with a bsis for examination of their approach to management. The Ferratt and Short instrument is presented in table 1.

Table 1: the Ferratt and Short survey instrument

1	meaningful work (performing important work, using several of my skills)
2	supportive relationships (working with friendly supportive people)
3	respect (feeling that my employer values me as a person and employee)
4	appreciation (receiving recognition or rewards for my work efforts)
5	clear job (having a clear understanding of what I am to do)
6	authority (having authority to make important decisions in my work)
7	feedback (receiving regular feedback on how I am doing my job)
8	influence (being able to influence what goes on in the department and company
9	high expectations (knowing my superiors expect me to do my best)
10	symphatetic understanding (having a supervisor that tries to understand me and my concerns)

In the original instrument by Ferratt and Short, respondants were asked to select the top five relevant factors. In order to be able to compare our results, it was decided to follow this approach here as wel.

However, after studying the model presented by Goldstein and Rockart, it was decided that some of the variables introduced there would provide a usefull addition to the factors suggested by Ferratt and Short. Table 2 shows an overview of the factors used by Goldstein and Rockart, the relationship between these factors and those suggested by Ferratt and Short, and finally the factors that were selected to be used in our survey instrument. For ease of use some simplifications were carried out here.

Simply adding these factors to the Ferratt and Short list would mean that the results obtained would not be comparable to the original data by Ferratt and Short. It was therefore decided to add these factors seperately. We decided not to ask respondents to select the top 2 or top 3, but to score them in an approach similar to that of Goldstein and Rockart on a five point likert scale. This similarity again allows comparison of the results obtained in our research with those obtained by the original research. Also, since ease of use was an important requirement, the instrument contained the main factors directly, instead of the fairly large numbers of underlying components that were identified by Goldstein and Rockart.

Table 2: factors used from the Goldstein and Rockart instrument

#	**Factor according to Goldstein and Rockart**	**Links to factor from Ferratt and Short**	**Factor added to instrument**
1	Skill variety		Skill variety
2	Task identity		Task identity
3	Task significance	1	Task significance
4	Autonomy	6	Autonomy
5	Feedback from job	7	Feedback from job
6	Role ambiguity		Role ambiguity
7	Role conflict		Role conflict
8	Person–role		
9	Intrasender		
10	Intersender		
11	Peer support	2	n.a.
12	Peer goal emphasis	2	
13	Peer work facilitation	2	
14	Peer interaction facilitation	2	
15	Supervisor support	2	
16	Supervisor goal emphasis		Supervisor goal emphasis
17	Supervisor work facilitation	3	Supervisor facilitation
18	Supervisor interaction facilitation	3	

The resulting two part instrument was translated into Dutch, and for each explanatory definitions where added that fitted in with both target groups. For example, for the factor ‘respect’, the following definitions were provided:

- for the analyst: have the feeling that my employer respects me as a person and as an employee,
- for the IT manager: express respect and appreciation to the analyst

Results

The resulting instrument was sent out (by email) to all 35 IT analysts and 8 IT managers from the business unit that participated in the study. The respons was 21 (60%) for the analysts (but one respons was found to be unusable) and 7 (88%) for the IT managers. The high response for the managers can be explained both by their interest in the reseach project as well as by some additional individual prompting. The results have been summarised in tables 3 and 4 below.

Table 3: Results I: factors derived from Ferratt and Short

Factor	Ranked by:			Agreement between:	
	Original	analysts	managers	Analysts and original	Analysts and managers
Meaningful work	1	1	1	+	+
Supportive relationships	2	2	10	+	-
Appreciation	4	3	6	+	0
Feedback	7	4	3	0	+
Authority	6	5	9	+	-
Respect	3	6	4	0	+
Sympathetic understanding	10	7	7	0	+
Clear job	5	8	5	0	0
Influence	8	9	2	+	-
High expectations	9	10	8	+	+

+: a difference of 0, 1, or 2
0: a difference of 3 or 4
-: a difference gt 4

Table 4: Results II: factors derived from Goldstein and Rockart

Factor	Ranked by:			Agreement between:	
	Original	analysts	managers	Analysts and original	Analysts and managers
Skill variety	4	1	9	0	-
Role ambiguity	1	2	2	+	+
Autonomy	2	3	8	+	-
Feedback from job	6	4	3	+	+
Task significance	5	5	5	+	+
Supervisor facilitation	3	6	6	0	+
Supervisor goal emphasis	8	7	1	+	-
Role conflict	7	8	4	+	-
Task identity	9	9	7	+	+

+: a difference of 0, 1, or 2
0: a difference of 3 or 4
-: a difference gt 4

Discussion

If we look at the comparison between the perceptions of the IT analysts participating in this study, and the results obtained previously we see that to a large degree there exists agreement as to the relative importance of the factors.

If we look at what IT analysts feel to be important, and what IT managers claim to consider in their daily management activities sime striking differences can be noted. A number of factors were either dwnplayed or over emphasised. These results were discussed in a meeting of IT management. It was felt by those involved that these results warranted a rething of the way they handled their affairs. A such, the study was considered to be a success by the department.

Given the limited number of participants extending the results of this survey should be done with some reservations. However, that large degree of agreement between the results reported by IT analysts and the resultes reported previously in literature, combined with the acceptance of the results by the management involved in the study would suggest that this instrument could be used in other environments. One might on the basis of these results even consider to accept the results from literature as a proxy for locally generated data by IT analysts. This would allow one to confine the survey to management only. This would no doubt speed up the process of data gathering but this might be at the expense of acceptance of the resulting conclusions.

References

Boehm, B.W., Ellis Horowitz, Ray Madachy, Donald Reifer, Bradford K. Clark, Bert Steece, A. Winsor Brown, Sunita Chulani, Chris Abts
Software Cost Estimation with Cocomo II,
Prentice-Hall, 2000.

Ferratt, T.W. and Larry E Short
Are information systems people different: An investigation of motivational differences,
MIS Quarterly, Volume 10 , Issue 4 (December 1986), pp. 377 – 388, 1986.

Couger, J.D and Zawacki R.A.,
Motivating and Managing Computer Personnell,
John Wiley & Sons, New York 1980

Goldstein, David K, Rockart, John F.,
An Examination of Work-Related Correlates of Job Satisfaction in Programmer/Analysts,
MIS Quarterly, Vol. 8, No. 2, pp. 103-105, 1984.

Fitz-enz
Who is the DP professional?
Tutorial IEEE Computer Society Press, 1978

Hackman, J.R. en Oldham, G.R.,
Motivation through the design of work: Test of a theory,
Organizational Behavior and Human Performance, vol. 16, 1976, pp. 250-279.

Herzberg, F.B. Mausner en B. snyderman,
The motivation to work.
New York: Wiley and sons, 1959

Sullivan, S.L.,
How much time do software professionals spend communicating?
ACM SIGCPR Computer Personnel**,** Volume 11 , Issue 4 (September 1988), pp. 2-5, 1988.

Managing Model Quality in UML-based Software Development

Christian F.J. Lange, Michel R.V. Chaudron
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513
5600 MB Eindhoven, The Netherlands
{C.F.J.Lange,M.R.V.Chaudron}@tue.nl

Abstract

With the advent of UML and MDA, models play an increasingly important role in software development. Hence, the management of the quality of models is of key importance for completing projects succesfully. However, existing approaches towards software quality focus on the implementation and execution of systems. These existing quality models cannot be straightforwardly mapped to the domain of UML models as source code and models differ in several essential ways (level of abstraction, precision, completeness and consistency). In this paper we present a quality model for managing UML-based software development. This model enables identifying the need for actions for quality improvement already in early stages of the life-cycle. Early actions for quality improvement are less resource intensive and, hence, less cost intensive than later actions. We discuss our experiences in applying the quality model to several industrial case studies. Finally we present a tool that vizualizes our quality model. This tools helps in relating management level quality data to detailed data about specific quality subcharacteristics.

I. Introduction

Models play a central role in modern software development. The Unified Modeling Language (UML) [16] has become the de facto standard for modeling software architectures and designs. For this reason we focus on UML as our main modeling language.

In the early stages of software development UML models are created in a sketchy manner. In this stage models serve as a means for understanding the problem domain and for communicating design decisions. The models are extended, refined and detailed throughout subsequent phases of the systems life-cycle. Ultimately, the UML models are used as a basis for implementation and testing. If a project uses poor models, it risks encountering problems such as misunderstanding [**?**], building the wrong product, large testing effort and ultimately a low quality product.

To reduce the risk of these problems occuring, models should be subject to continuous quality control - as is common for the case of source code. There are however, significant differences between source code and models. Source code closely corresponds to the executing systems as the latter can be automatically derived from the former. This correspondence is much weaker for models. Generally, models are more abstract than sources; i.e. details are left out. Furthermore, source code is subject to a formal syntax that is enforced by a compiler. UML modeling tools enforce syntactical constraints very weakly (generic drawing tools not at all) and semantic constraints are not enforced at all. Thus models may be incomplete and inconsistent. The inconsistency is aggrevated by the fact that models employ multiple views. Inconsistencies between views are seldomly identified. Summarizing, UML and source code differ in abstraction level, precision, completeness, consistency and correspondence to the ultimate system (depicted in Table I).

We based these observations on a number of industrial case studies into the use of the UML ([11]). In these case studies we observed that the UML is used in many different ways, that models range from sketchy to detailed, but generally contain a large amount of defects.

To a degree these deficiencies seem inherent in UML models as it is a notation that aims to bridge the spectrum from informal problem statement to detailed design. However, these deficiencies potentially endanger the smooth execution of a project. Hence there is a need for objective means for establishing the quality of UML models.

These observations motivated us to develop a quality

model for UML models that takes into account the aforementioned characteristics of models as well as the manners and phases in which an organization aims to use the models. The quality model can be seen as complementary to existing quality model that focus on the quality of the system. This quality model for UML models aims to enable identifying the need for actions for quality improvement already in early stages of the life-cycle. Early actions for quality improvement are less resource intensive and, hence, less cost intensive than later actions [2]. Additionally new model-centric development techniques such as MDA [15] require high-quality models.

This paper is structured as follows. In Section II we summarize the most important approaches to software quality. In Section III we present our quality model for UML models. In Section IV we present our experiences and ideas in applying the quality model. Section V concludes and discusses directions of ongoing and future work.

II. Software Quality

In this section we review the literature on software quality and relate our approach for quality of UML models to it.

A. Perspective on Quality

The topic (software) product quality is frequently discussed, and most organizations and stakeholders agree that quality is an important property of products. But there are many definitions of quality. To achieve the common goal *quality*, one first has to agree upon the same definition. Garvin [7] defines five views of product quality:

- the transcendental view: quality is seen as something than can be perceived but not defined.
- the user view: quality is seen as the fitness for the user's purpose.
- the manufacturing view: quality is seen as conformance to the product's specification.
- the value-based view: quality is seen in terms of a product's potential to generate economical turnover.
- the product view: quality is decomposed into several characteristics that are inherent to the product.

The main application of our quality model is during development of the product. In this phase neither the user's perception nor the product's economical potential can be measured. During development only internal properties of the product and its model can be measured, hence, we apply the product view of quality since we decompose quality into characteristics that are inherent to the product.

	Source Code	UML Model
Abstraction	low	high - medium
Precision	high	medium - low
Completeness	high	low
Consistency	high	medium - low

TABLE I. Differences between Source Code and UML Models

B. Existing Approaches

Early quality models that still serve as reference models are the models proposed by Boehm [3] and by McCall [13], the ISO 9126 [8] standard for software quality contains a quality model that is based on McCall's model. These models together with the model used by Rombach [18] also decompose quality in subcharacteristics that are inherent to the product.

A prominent way of defining subcharacteristics is by means of metrics. In literature there are many metrics [6] defined that are based on source code (e.g. Chidamber and Kemerer's suite [4] or McCabe's Cyclomatic Complexity Measure [12]). The quality of a system is assessed using metrics by relating metrics to quality attributes as described in a *quality model*.

The models by Boehm and McCall structure the subcharacteristics into three hierarchical levels, where each level represents a different class of characteristics. The three levels of McCall contain *Uses*, *Factors and Criteria*, respectively. Boehm uses different vocabulary but similar meanings. The levels in Rombach's model are only hierarchically ordered and do not denote different classes of characteristics. We also use the three-level approach with a specific class of characteristics for each level in our quality model.

The mentioned quality models concern a system during its entire life-cycle. Besides the product view they take also the user view and the manufacturer view into account. These models have a rather general view on quality with an emphasis on the product in operation and in maintenance. Both, the Boehm and the McCall model define three *Uses* as in the highest level of the quality tree: *operation, maintenance and transition*. (Note that Boehm and McCall use different terminology for some concepts. Boehm defines transition as *portability* and locates it in the intermediate level.)

The existing quality models lack the distinction between the system and its description. In the next section we introduce a new concept at the top level of the quality tree to include this distinction in our development-centric quality model.

III. Quality Model

In this section we describe the specific needs for a quality model that focusses on model-based development and we present the quality model and its concepts.

A. Need for a model-centric quality model

Programming a software system means writing its source code. Nowadays most source code is written in high-level programming languages like Java or C++. The programming languages have a formal syntax and semantix and can automatically be transformed to machine readable instructions. Hence, the artefact source code (being the description) and the system (the product) coincide. Therefore the description describes the product unambiguously. The dominant quality models described in Section II do not distinguish between quality characteristics that are inherent to the system and quality characteristics that are inherent to its description.

Models describe systems on at higher abstraction levels than source code. Additionally the UML introduce degrees of freedom and potential for inconsistency. For example the UML lacks a formal semantics and does not have strict guidelines which of the 13 diagram types and their elements must be used to describe a specific purpose. This results in the fact that a model does not describe the system unambiguously. Unlike for source code, there is a gap between the artefact *model* and the system described by it.

Models are extensively used during development, but also during maintenance. Hence, there is a need to describe the quality characteristics of models. The quality model presented in this section distinguishes between quality characteristics of the model as a description and the system described by it.

B. Concepts in this model and their definitions

In this section we describe the levels of our quality models and the concepts that belong to the levels. **Level 1: Use.** The top level of our quality model describes the high-level use of the artefact. The well known quality models described in Section II define three uses at this level:

- **Operation**: this use combines quality characteristics that concern the product when it is executing. This implies that the system is readily implemented. The characteristics concerning the system's operation relate to the user view of product quality. We focus on the product view, hence the operation use is out of the scope of our quality model.
- **Transition**: this use combines quality characteristics that concern the product when it is moved to another environment. Transition issues are implementation dependent and are not addressed by modelling in the design phase of a system. Therefore the transition use is out of the scope of our quality model.
- **Maintenance**: this use combines quality characteristics that concern the product when it is changed.

These uses do not addresses development of a model-based design of the system. Therefore we introduce a new use:

- **Development**: this use combines quality characteristics that concern the product and its artefacts in the phases before the product is finished.

UML models are used in the development and maintenance phases. Therefore our quality model takes these two uses into account. The decomposition of these uses is presented in the sequel.

Level 2: Purposes of Modeling. The second level of the quality model contains the *purposes* of the artefact. A purpose describes why the artefact is used. Usually an artefact is not used for all purposes at the same time (in fact, all purposes *cannot* be served at the same time; see Section IV-A).

The purposes for modeling are presented and described in Table II.

Level 3: Characteristics. The third level of our quality model contains the inherent characteristics of the artefact. The characteristics described in Table III. According to the distinction between characteristics of the model and characteristics of the system (see Section III-A) we indicate for each characteristic whether it is a model characteristic (column *M*) or a system characteristic (column *S*). Some characteristics are defined for both model and system. The only exception is *correspondence*: this is a characteristic of a model-system pair. Correspondence between a UML model and the source code of a system is needed to enable using the model to understand the system.

Level 4: Metrics and Rules. The characteristics concepts of the three aforementioned levels of the quality model cannot be measured directly from the artefact. The characteristics can be measured by (a combination) of metrics and rules. A metric is a mapping from the empirical domain to the numerical domain, such that the its value reflects the level of some property of the artefact or an element of the artefact. Rules can be seen as special cases of metrics. They are mappings to a binary value: true or false. Rules are usually defined for elements of artefacts (e.g. 'Abstract class X must have a subclass'). Hence, a rule is a numeric metric for an entire artefact (i.e. the number of elements violating the rule).

A large number of metrics and rules are proposed in literature (e.g. [6], [9], [4]). For the sake of brevity we cannot present all metrics in this paper. We present a representative set of rules and metrics that are applicable in our quality model. The presented list contains some

Purpose	Description
Modification	The model and the system enable easy modification. Possible modifications are corrections, i.e. removal of errors, extensions of the system, or adaptive changes due to changed requirements.
Testing	The model is used to generate test cases.
Comprehension	The model and the system are easily comprehensible, i.e. the system elements and their functionality are modelled such that they can be understood correctly in a reasonable amount of time
Communication	The model enables efficient communication about the system's elements, their behavior and the design decisions. Communication includes communication during the development phase with different stake holders and documentation for understanding the system in later phases such as maintenance.
Analysis	The purpose of the model is to explore and analyse the problem domain including its key concepts and making some early design decisions.
Prediction	The model is used to make predictions about quality attributes of the eventual implemented system.These predictions are used to make early and therefore cheaper improvements of the architecture and design.
Implementation	The model is used as basis for (manual) implementation of the source code of the system.
CodeGeneration	The model is used to automatically generate the source code of the system. Code generation can be complete or partial (only a skeleton of the source code is generated).

TABLE II. Purposes of modelling

Characteristic	Model	System	Description	Ref
Complexity	√	√	Complexity: measures the effort required to understand a model/system	[6]
Traceability	√	√	The extent that relations between design decisions are explicitly described.	
Modularity		√	The extent that its parts are systematically structured and seperated such that they can be understood in isolation.	
Completeness	√	√	A system possesses completeness to the extent that it's functionality covers all requirements. A model possesses completeness to the extent that overlapping parts of different views contain the same elements and the system is completely described by the model.	[11]
Consistency	√		The extent that no conflicting information is contained	[5]
Communicativeness		√	The extent that it facilitates the specification of inputs and provides outputs whose form and content are easy to assimilate and useful.	[3]
Self-Descriptiveness	√	√	The extent that it contains enough information for a reader to determine its objectives, assumptions, constraints, inputs, outputs, components, and status.	[3]
Detailedness	√		The extent that it describes relevant details of the system	
Balance	√	√	A model possesses balance to the extent that all parts of the system are described at an equal degree of all other model characteristics. A system possesses balance to the extent that all parts of it are equally with respect to all other system characteristics	
Conciseness	√		The extent that the system is described to the point and not unnecessarily extensive.	[3]
Esthetics	√		The extent that its graphical layout enables ease of understanding of the described system.	[17]
Correspondence	√	√	A pair of model and system possess correspondence to the actual system to the extent that system elements, their relations and design decisions are the same in the model and the system	

TABLE III. Characteristics

specific metrics and rules, but also families of related metrics and rules. The metrics and rules of models are presented and described in Table IV. The relation between metrics and rules and the characteristics of the quality model is explained in the sequel.

C. Relations between Concepts

Now we are ready to present the relations between the aforementioned concepts. The concepts together with the relations between them form the actual quality model. Figure 1 shows the quality model. The arrows indicate relations between two concepts of different levels. An arrow indicates that the concept of the lower level influences the concept of the higher level. The arrows can also be interpreted as follows: a lower level concept *is part of* all higher level concepts to which it is related by an arrow, and a higher level concept *contains* the related lower level concepts.

The relations between metrics and rules are shown in Table V

IV. Applying the Quality Model

We have explained the quality model for UML models. This section describes how the quality model fits into different phases of the development. Additionally we present a tool prototype that we have implemented to support working with the quality model. We report findings from industrial case studies.

Name	Description	Ref
Dynamicity	Measures the complexity of a class' internal behavior. Dynamicity is based on the assumption that message calls correspond to state transitions. It takes information from the interaction diagrams into account.	[10]
Ratios	Ratios between number of elements (e.g. number of methods per class.)	
DIT	Depth of Inheritance Tree	[4]
Coupling	The number of other classes a class is related to.	[4]
Cohesion	measures the extent to which parts of a class are needed to perform a single task	[4]
Class Complexity	the effort required to understand a class	[14]
NCU	Number of classes per use case.	[14]
NUC	Number of use cases per class.	[14]
Fan-In	The number of incoming association relations of a class. Measures the extent to which other classes use the class' provided services.	
Fan-Out	The number of outgoing association relations of a class. Measures the extent to which the class uses services provided by other classes.	
Naming Conventions	Adherence to naming conventions	
Design Patterns	Adherence to design patterns	[1]
NCL	Number of crossing lines in a diagram.	[17]
Multi defs.	Multiple definitions of an element (e.g. class) under the same name.	
ID Coverage	Interaction diagram coverage. This is a family measures the extent to which the interaction diagrams cover the elements described in the structural diagrams. Examples are percentage of classes instantiated in interaction diagrams and percentage of use cases described by interaction diagrams.	[10]
Message needs Method	This consistency rule states that each message in an interaction diagram must correspond to a method defined in the class diagram.	
Code Matching	This family of metrics measures the extent to which the code matches the model. Example: percentage of model classes that occur in the code.	
Comment	Measures the extent to which the model contains comment. Example: lines of comment per class.	

TABLE IV. Metrics and Rules

Metrics and Rules	Modularity	Complexity	Completeness	Consistency	Communicativeness	Self-Descriptiveness	Detailedness	Balance	Conciseness	Esthetics	Correspondence
Dynamicity		√	√					√			
Ratios	√		√				√	√	√		
DIT	√	√			√			√	√		
Coupling	√							√			
Cohesion	√	√						√			
Class Complexity								√			
NCU	√	√						√	√		
NUC	√	√						√	√		
Fan-In	√							√			
Fan-Out	√							√			
Naming Conventions						√		√			
Design Patterns	√		√			√		√	√		
Layout-Guidelines								√		√	
Multi defs.				√				√			
ID Coverage			√					√			
Message needs Method			√	√				√			
Code Matching			√					√			√
Comment						√		√			

TABLE V. Relations between metrics and rules and characteristics

A. Tailoring the quality model for different phases

The quality model covers eight purposes of modelling. The need for a specific purpose is not the same for every situation. When using the quality model, the important purposes, and, hence the important characteristics, should be carefully selected. The importance of purposes depends on the project phase, the development process and characteristics (and taste) of the development team. For example in a development team that is locally distributed over different countries, comprehension and communication are important purposes. In a project that develops a very

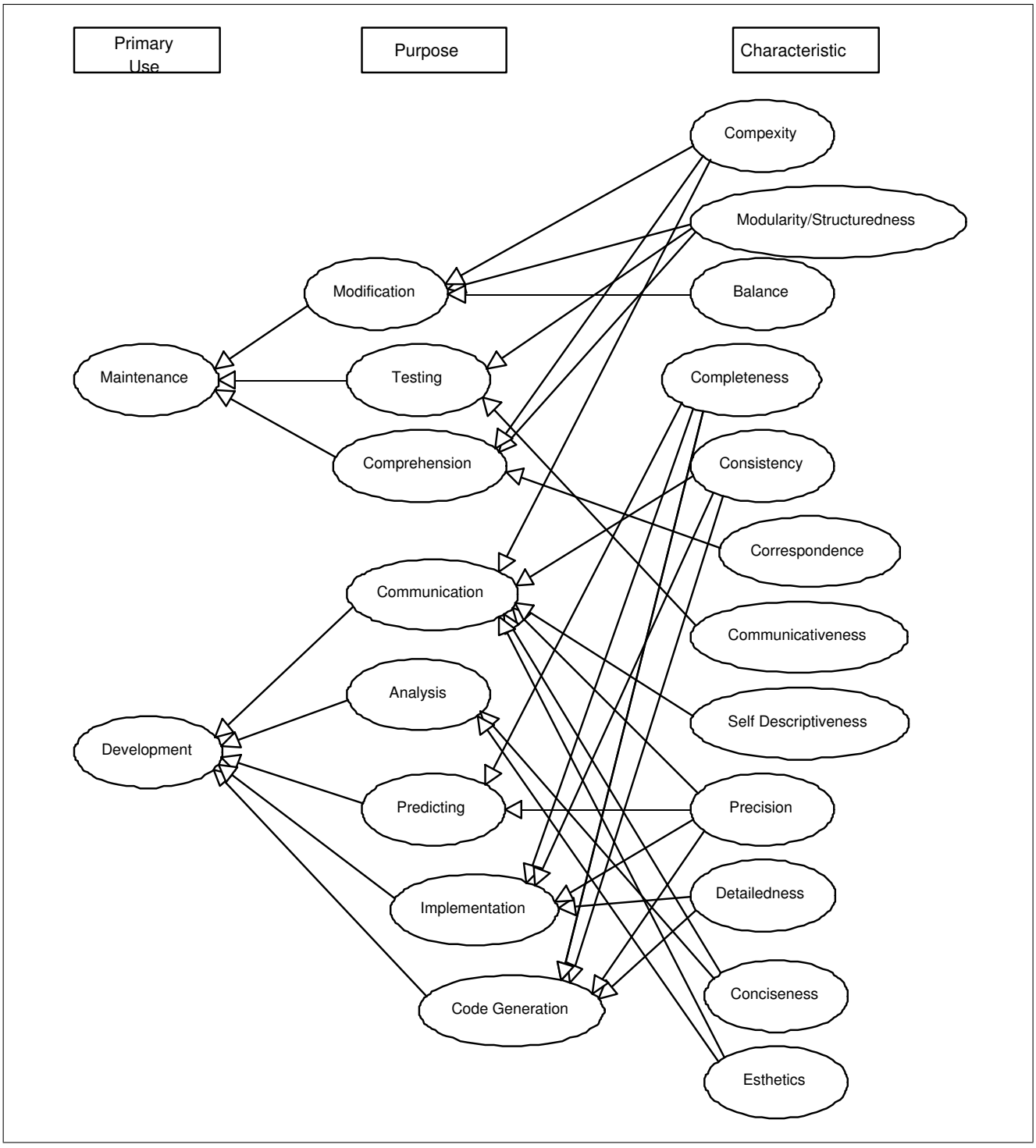


Fig. 1. Quality Model

large system containing many risks, early prediction of the quality attributes of the system is important to enable cost-effective improvements.

In several industrial case studies (see Section IV-B) we have discussed importance of modelling purposes with project members. As a result, we have have built a schema that relates purposes of modelling to project phases. Figure 2 shows the relations between project phases and

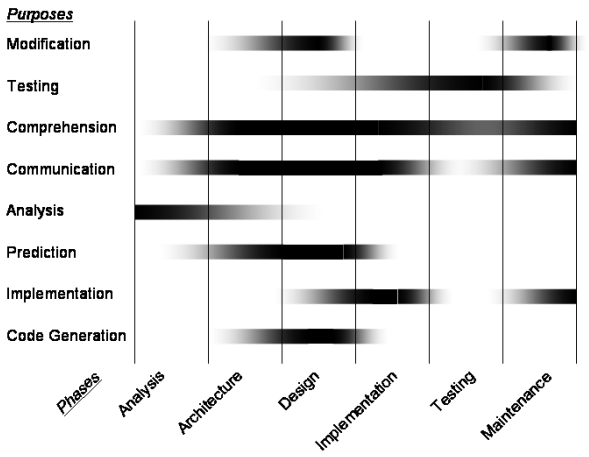


Fig. 2. Relations between modelling purposes and project phases

modelling purposes. The shaded boxes show that the purpose is seen as important for the specific phase. Darker shading indicates higher importance. This schema can be seen as an indication, but different priorities for individual projects are of course possible.

B. Experiences from industrial case studies

In the scope of our research project we conduct case studies with industrial partners for three reasons: to empirically validate the developed methods, to get direct feedback on their practical usefulness and to get insight in problems that occur when using the UML in industrial practice. A typical case study is structured in three steps:

1) **Interview.** In an initial interview with a representative of the project team we assess the project context. We investigate the phase, the purpose of modelling, the size (in persons and man years), the development process, known problems and other characteristics of the project.
2) **Analysis.** We analyze the UML model(s) using our metrics and rule-checking tools. The raw results are related to the described quality model
3) **Feedback.** The analysis results and findings are presented to the project team and the team members have the opportunity to discuss and interpret the findings.

Here we present the characteristics of a selection of our case studies including the project phase and the purposes of modelling in the project. The case studies were conducted in five different companies with different application domains.

Table VI shows the data from the case studies including model size in terms of number of classes, team size, phase and purposes of modelling indicated by the project team (marked by a '$\surd$'). Each project team chose an individual set of modelling purposes. The analysis was tailored according to the indicated purposes.

The analysis of the case studies discovered the following concrete problems in the UML models:

- *Dead model parts*. In source code 'dead code' means regions of the source code that are outdated, but are still part of the artefact. In UML models we found similar problems. During our analysis we found complete packages and sometimes parts of packages of model that were not used any more and that contained outdated information. In the case analyzed models outdated regions of the model were neither removed nor explicitly marked as outdated. This can cause misunderstanding of the model and unnecessary effort for reviewing the model. Poor values for the characteristics *conciseness* and *consistency* indicate this problem. This problem was found in case studies Case A and Case E.
- *Wrong model checked in*. In Case D the characteristics *completeness* and *consistency* showed very low values. In particular the ID coverage metric and rules for consistency between sequence diagram and class diagram showed that there were issues with the sequence diagrams. Due to these results the responsible developer discovered, that he had uploaded a premature version of the model (with scratches of the sequence diagrams) to the configuration management system. The correct version was only on his local machine and not accessible by the other team members.
- *Large number of inconsistencies*. All case studies showed large numbers of consistency defects as reported in [11]. For some case studies (Case G and Case H) we have analyzed subsequent versions of the model. We observed that the number of inconsistencies decreased when the developers started using the proposed methods for monitoring the quality of their systems.
- *Disbalance between developers*. In Case B the value for the characteristic *disbalance* was low. The developers had large differences in their habits of modelling, with respect to their modelling style, level of detail of modelling, and types and amount of inconsistencies. This is a risk when the model is used for communication between different developers.

All of the mentioned problems were unrecognized by the developers before the case study. The developers acknowledged the detected problems.

Case	# of Classes	# of Developers	Project Phase	Modification	Testing	Comprehension	Communication	Analysis	Prediction	Implementation	Code Generation	Remark
Case A	734	5	Maintenance				√			√		
Case B	443	10	Implementation			√	√			√		
Case C	4	3	Design					√		√		
Case D	51	6	Architecture			√		√				
Case E	705	6	Design	√	√		√		√	√		detailed version of Case D
Case F	14	2	Analysis			√		√				
Case G	46	2	Design						√	√	√	detailed version of Case F
Case H	73	1	Design			√	√					simplified version of Case I
Case I	359	5	Design	√			√		√	√		

TABLE VI. Data form industrial case studies

C. QualityView: Tool-support for quality modelling

Information about a product's quality is useful for different stake holders. Managers use the information for example to monitor progress, developers use the information to identify flaws in the product and find opportunities for improvement. The most prominent sources of information about a the quality of a system under development are metrics tools such as SDMetrics [19]. These tools deliver fine-grained information at the class- or system-level. This information is at the lowest level of quality models like McCall, Boehm and our work. This data must be manually related to the higher level concepts of the quality model like characteristics, purposes and uses.

We have developed a prototype tool to assist the user in relating metrics data to quality attributes. The QualityView tool visualizes the entire quality model as a tree. The user can also tailor the quality model such that only the purposes of importance are visualized or extend the quality model by adding concepts, levels and relations. Besides allowing the user to tailor the quality model to his needs the tool provides two use cases in its current version:

- *Assistance in navigating through the quality model*. The user can select a concept at any level of the quality model. All related concepts are highlighted to improve the understanding of how the concepts influence each other. A screenshot of this functionality is shown in Figure 3
- *Visualizing the values for the concepts*. QualityView is built on top of a metrics tool for UML models and takes the metrics as input. The value for each concept in the quality model is visualized in a three-dimensional view of the quality model: the height of the bar indicates the extent to which the analyzed model fulfills the concept. The value for concepts is composed of lower-level concepts. The composition function can be defined by the user to fit his specific needs. A screenshot of the three-dimensional visualization of values for concepts is shown in Figure 4.

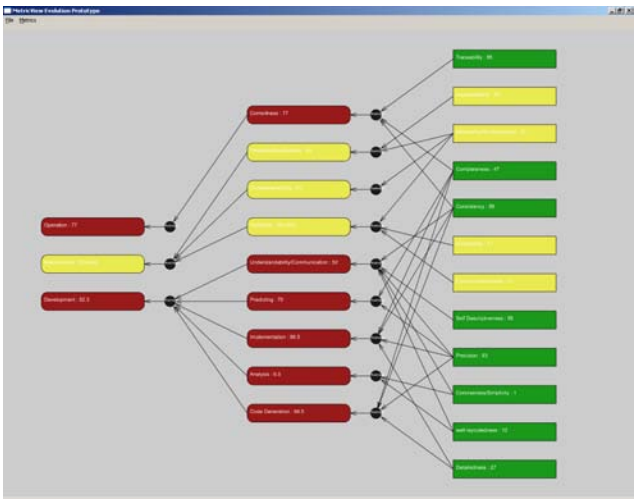


Fig. 3. Visualization a selected concept and all related concepts using QualityView

Note to the reviewers: The submission deadline did not permit us to include screenshots of visualizations of data from industrial case studies. We will include screenshots of data from industrial case studies in the final version of this paper.

V. Conclusions and Future Work

Models play an increasingly important role in software development. In order to support the management of quality from early phases of architecture and design, techniques are needed for assessing quality of models. Based on a number of industrial case studies, we have proposed a

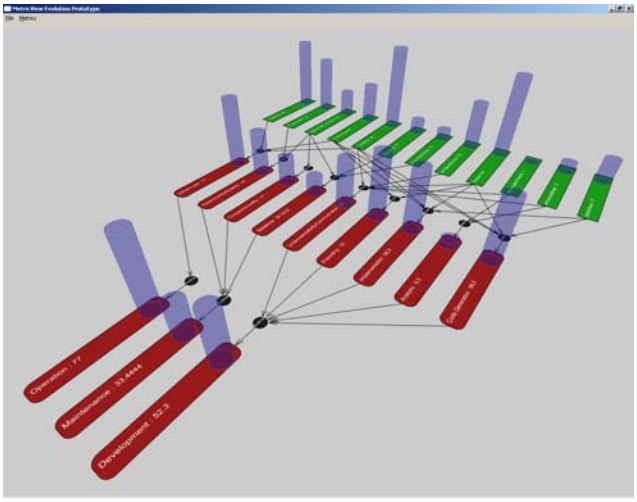


Fig. 4. Three-dimensional visualization of the values for each concept in the quality tree using QualityView

quality model for UML models. This model takes into account the different uses of models in a project as well as the phase in which a model is used. This model enables identifying the need for actions for quality improvement already in early stages of the life-cycle.

The main contribution of this paper is that we have developed a quality model that takes into account quality characteristics of the model as well as quality characteristics of the system. Our quality model is organized in a three-level decompositional structure. On the highest level we have introduced a new use: development. In the second level of our quality model we propose purposes for modeling. We relate these purposes to different phases in the life-cycle of the product. The set of important purposes differs per project phase, such that the quality model can be tailored according to the purposes of the current phase. The lowest level of the model is related to a set of metrics and rules that measure the properties inherent to UML models.

We have developed a tool on top of a software metrics tool for UML models. The tool visualizes the quality model as a tree and supports navigating through the tree. Additionally the tool takes the metrics data as input and intuitively visualizes the value for each concept of the quality model. When evaluating a model with respect to quality attributes this visualization assists in identifying strong and weak quality attributes of the model.

In further studies the relations between metrics and rules and the quality characteristics as well as between the different layers of the quality model should be empirically investigated. Future work also includes studying the evolution of quality over time, i.e. over different versions of a model and of the one model in different phases.

Acknowledgements. We thank Martijn Wijns for implementing the prototype of the QualityView tool. Additionally we thank our industrial partners for sharing their data with us to conduct case studies and for useful discussions.

References

[1] *Design Patterns: Elements of reusable Object-Oriented Software*. Addison Wesley, 1994.

[2] Barry Boehm. *Software Engineering Economics*. Prentice Hall, October 1981.

[3] Barry W. Boehm, John R. Brown, Hans Kaspar, Myron Lipow, Gordon J. Macleod, and Michael J. Merrit. *Characteristics of Software Quality*, volume 1 of *TRW Series of Software Technology*. North-Holland Publishing Company, Amsterdam, 1978.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[5] Steve Easterbrook and Bashar Nuseibeh. Using ViewPoints for inconsistency management. *BCS/IEE Software Engineering Journal*, pages 31–43, January 1996.

[6] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics, A Rigorous and Practical Approach*. Thomson Computer Press, second edition, 1996.

[7] David Garvin. What does 'product quality' really mean? *Sloan Management Review*, 26(1):25–45, 1984.

[8] ISO/IED FCD 9126-1.2. *Information Technology - Software Product Quality*, part i: quality model edition, 1998.

[9] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley Professional, 2nd edition, September 2002.

[10] Christian F. J. Lange. Empirical investigations in software architecture completeness. Master's thesis, Technische Universiteit Eindhoven, September 2003. No. 969.

[11] Christian F. J. Lange and Michel R. V. Chaudron. An empirical assessment of completeness in UML designs. In *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering (EASE'04)*, pages 111–121, 2004.

[12] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[13] J. A. McCall, P. K. Richards, and G. F. Walters. *Factors in Software Quality*, volume vol 1-3 of *AD/A-049-015/055*. Springfield, 1977.

[14] Johan Muskens, Christian F. J. Lange, and Michel R. V. Chaudron. Experiences in applying architecture and design metrics in multi-view models. In *Proceedings of EUROMICRO 2004, Rennes, France*, August 2004.

[15] Object Management Group. *MDA Guide, Version 1.0.1*, omg/03-06-01 edition, June 2003.

[16] Object Management Group. *Unified Modeling Language, Adopted Final Specification, Version 2.0*, ptc/03-09-15 edition, December 2003.

[17] Helen C. Purchase, Linda Colpoys, Matthew McGill, David Carrington, and Carol Britton. UML class diagram syntax: an empirical study of comprehension. In *Australian symposium on Information visualisation*, volume 9, pages 113–120, September 2001.

[18] H. Dieter Rombach. *Quantitative Bewertung von Software-Qualitäts-Merkmalen auf Basis struktureller Kenngrössen*. Phd thesis, Universität Kaiserslautern, June 1984.

[19] Jürgen Wüst. The software design metrics tool for the UML. http://www.sdmetrics.com.

A FRAMEWORK FOR THE V&V CAPABILITY ASSESSMENT FOCUSED ON THE SAFETY-CRITICALITY

Kyung-A Yoon[†] *Seung-Hun Park*[†] *Hoon-Seon Chang*[‡] *Jae-Cheon Jung*[‡] *Doo-Hwan Bae*[†]

[†] Division of Computer Science
Department of EE and CS
KAIST, Korea
kayoon,seunghun,bae@se.kaist.ac.kr

[‡] Korea Power Engineering Company Inc.,
Korea
jcjung,hschang@kopec.co.kr

ABSTRACT

As the importance of verification and validation(V&V) activities is growing, the necessity of criticality-based assessment framework of V&V capability is increasing in the safety-critical software organizations. Although several maturity models for quality improvement such as Capability Maturity Model Integration(CMMI), Testing Maturity Model(TMM), etc.were developed following the industry's request, they have the limitation to support enough V&V assessment for the safety-critical software. In this paper, we propose the framework of V&V capability assessment in order to assist in performing "safety-criticality" assessment. To provide the essential V&V practices and the confidential capability level scheme, our framework has five V&V capability levels based on the integrity level scheme of IEEE Std.1012 and the V&V tasks that come from IEEE Std.1012, 7-4.3.2, 1228, RG 1.168, and CMMI.

1. INTRODUCTION

In the safety-critical systems such as Nuclear Power Plant (NPP) systems, the very high confidence for software quality is required because those systems have high catastrophic potential and relatively low-cost alternatives [1]. Recently, the concept of software V&V is accepted as a way to assure the quality of the new digitalized safety-critical system. As the importance of V&V activities is growing, the necessity of criticality-based assessment framework of V&V capability is increasing in the safety-critical software organizations since they want to select the internal/external supplier which has enough V&V capability to develop safety-critical software. Several maturity models for quality and productivity improvement are developed following the industry's request. CMMI [2] is organized with five maturity levels and Process Area(PA)s to support process improvement. Although CMMI provides the V&V practices in its two PAs, Verification and Validation, it does not sufficiently address the V&V process. Therefore, TMM [3, 15] is developed to support to assess and improve a testing process. However, TMM does not focus on the safety-criticality since its target to apply is general software V&V organization.

In this paper, we propose the framework of V&V capability assessment focused on the safety-criticality in order to assist in performing safety-criticality assessment. The software integrity level and the minimum V&V tasks sets of IEEE Std.1012[1] [4] are used as the basis of our V&V assessment framework. In addition, V&V tasks which are recommended by IEEE Std.7-4.3.2 [5, 6], 1228 [7] and RG 1.168 [9] are included to make our framework reflect the safety-critical software domain. To complement our model, we add the V&V tasks that are identified by the mapping results between our V&V tasks and CMMI PAs, and ISO 9001 [10].

The remainder of this paper is organized as follows. Section 2 briefly explains CMMI, ISO 9001, TMM and IEEE Std 1012, 7-4.3.2, 1228, and RG 1.168 as background. The development steps, detailed information, and analysis of our framework are described in Section 3 and 4. Finally, we conclude with future works in Section 5.

2. BACKGROUND

In this section, we illustrate the maturity models and several standards which are referenced to develop our framework. The representative process assessment models are CMMI and ISO 9001. CMMI was developed in 1998 by Software Engineering Institute(SEI) of Carnegie Mellon University.

This work was supported by the Ministry of Information & Communication, Korea, under the Information Technology Research Center(ITRC) Support Program.

[1] To save the space, we abbreviate the published year which follows the standard number.

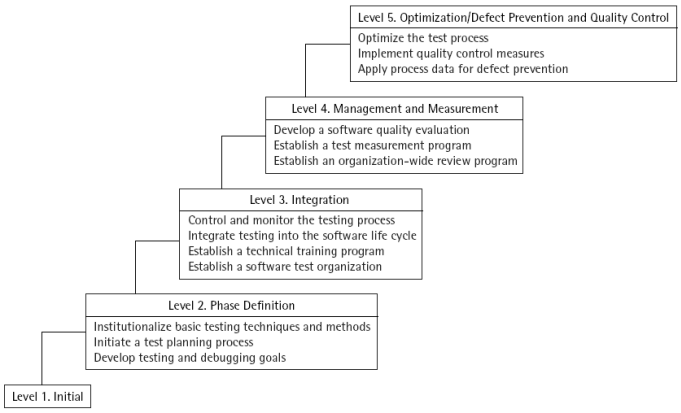


Figure 1: TMM maturity levels and goals.

It designed by combining SW-CMM, the Systems Engineering Capability Model(SECM), and the Integrated Product Development(IPD) CMM to support the Software Process Improvement(SPI) activities across the different disciplines in an organization and reduce the cost of performing a training, appraisals, and SPI activities [2]. ISO 9001:2000 specified requirements for a quality management system to assure the quality in design, development, production, installation and servicing. Although these two models are accepted by industry, they don't sufficiently address the software V&V process [10]. CMMI provides the Verification and Validation PAs in level3. However, they are not enough to provide sufficient visibility into the V&V capability. To assist software development organizations in evaluating and improving their testing processes, TMM was developed by Illinois Institute of Technology in 1996 [3, 15]. In TMM, "testing" is applied in its broadest sense to encompass all software quality-related activities [3]. As shown in Figure 1, TMM is characterized by five testing maturity levels that can be used in conjunction with the CMMI. However, TMM does not focus on the safety-criticality since its target to apply is a general software V&V organization. The limitation of TMM with the viewpoint of assessing the criticality-based V&V capability will be discussed in Section 4.

To develop our assessment framework, the criticality-based assessment framework of V&V capability, the specific V&V practices related to safety-critical software are essential components. In order to find these specific practices, we referred to IEEE standard 1012-1998, 7-4.3.2-1993 /2003, and 1228-1994, and NRC RG 1.168. IEEE Std 1012-1998, the international standard for software verification and validation, is a process standard that defines the V&V processes in terms of specific V&V activities and related tasks [4]. It defines four software integrity levels to describe the criticality of the software, the minimum V&V tasks to be assigned to each integrity level, and the optional V&V tasks to allow the user to tailor the V&V process. The criticality-based software integrity levels denote a range of software criticality values necessary to maintain risks within acceptable limits [4]. According to this standard, software criticality may include safety, security, software complexity, performance, reliability, or other characteristics. In relation to minimum V&V tasks and software integrity level, IEEE Std.1012 introduces the intensity and rigor applied to V&V tasks. Higher software integrity levels require the application of greater intensity and rigor to the V&V task. Intensity includes greater scope of analysis across all normal and abnormal system operating conditions and rigor includes more formal techniques and recording procedures. IEEE Std 7-4.3.2-1993, the standard criteria for digital computers in safety systems of nuclear power generating stations, presents tasks to support the specification, design, and implementation of computers in safety systems of nuclear power generating stations. Annex E of this standard describes V&V tasks throughout life cycle. IEEE Std 1228-1994, the standard for software safety plans, recommends safety-analysis actions in its Annex. The final standard that we referred to is RG 1.168 which developed by U.S.Nuclear Regulatory Commission in 1997. It describes a method of the V&V, reviews, and audits acceptable to the NRC staff for complying with parts of the NRC's regulations for promoting high functional reliability and design quality in software used in safety system.

3. FRAMEWORK FOR THE V&V CAPABILITY ASSESSMENT FOCUSED ON THE SAFETY-CRITICALITY AND ITS DEVELOPMENT STEPS

In this section, we give an explanation on the framework for the V&V capability assessment focused on the safety-criticality and the development steps of its capability model. Figure 2 presents our overall approach. Our framework consists of the Criticality-Based V&V Capability Model(CB-VVCM), the questionnaire, and the mapping table with other models. The CB-VVCM has five V&V capability levels based on the integrity level scheme of IEEE Std.1012 and the V&V tasks that come from IEEE Std.1012, 7-4.3.2, 1228, RG 1.168, and CMMI. The questionnaire is created using the CB-VVCM tasks and the mapping table is the results of mapping the CB-VVCM tasks to the CMMI PAs and the ISO 9001 clauses. The development steps of the CB-VVCM are illustrated in Figure 3.

3.1. Development steps of the CB-VVCM

3.1.1. Create the initial model by deciding the capability level scheme and assigning a task set

Deciding the capability level scheme is one of the important things in developing our framework. The capability level

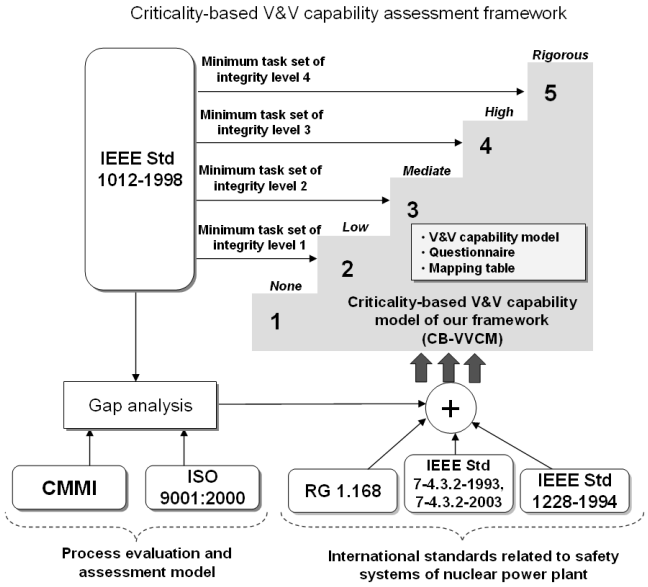


Figure 2: The overall approach.

should reflect well the software organization's V&V capability with the view point of "safety-criticality." As we mentioned in section 2, IEEE Std.1012 has the four criticality-based software integrity levels and the minimum V&V task sets for each of them. Although this integrity level is originally designed to perform the proper V&V tasks according to the assigned criticality degree of software, it can be used to assess the organization's V&V capability, too. Because critical, high-integrity software typically requires a larger set and more rigorous and intensive application of V&V tasks, the software organization which can observe the minimum V&V tasks of the high integrity level has the high criticality-based V&V capability. Therefore, the criticality-based V&V capability level of an organization is decided by assessing whether the organization performs its defined minimum V&V tasks or not. The relation between the software integrity levels of IEEE Std.1012 and our V&V capability levels is presented in Figure 2. Table 1 summarizes the objective of each criticality-based V&V capability level of CB-VVCM. After deciding the scheme of capability level and assigning the minimal task sets for each level, the initial CB-VVCM is created.

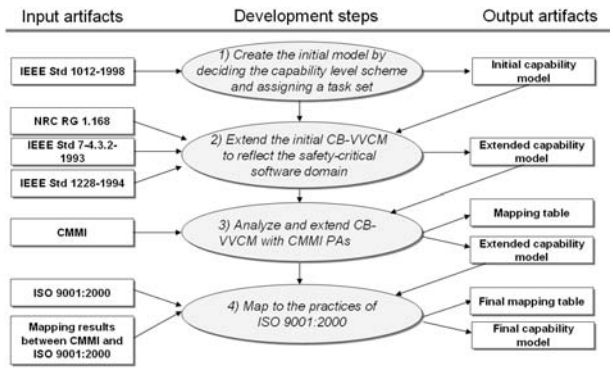


Figure 3: The development steps of the CB-VVCM.

Table 1: The objective of each criticality-based V&V capability level in CB-VVCM.

Level	Objectives
5 *Rigorous*	Prevent the critical software system failure that - Directly affects the loss of life - Don't permit any mitigation strategies is implemented
4 *High*	Prevent the software system failure that - Directly affects the major and permanent injury - Affects important system performance - Permits that workaround strategies is implemented to compensate partially for loss of performance
3 *Mediate*	Prevent the software system failure that - Directly affects the minor injury or illness - Affects system performance - Permits that workaround strategies is implemented to compensate completely for loss of performance
2 *Low*	Prevent the software system failure that - Creates inconvenience to the user if the function does not perform in accordance with requirements - Gives noticeable effect on system performance
1 *None*	

3.1.2. Extend the initial CB-VVCM to reflect the safety-critical software domain

The part of optional V&V tasks in IEEE Std.1012 which can support the safety-critical software V&V is not included to the minimum V&V tasks because the application target of IEEE Std.1012 is general software systems. Since the initial CB-VVCM has the same minimum V&V tasks set of IEEE Std.1012-1998, it should be strengthened by adding particular V&V tasks for the safety-critical software. To complement CB-VVCM, we consider the following standards which contain the safety-critical specific V&V tasks: IEEE Std.7-4.3.2, 1228, and RG 1.168. All V&V tasks which are recommended from these standards are part of the minimum V&V tasks and the optional V&V tasks of IEEE Std.1012. Except five optional V&V tasks, V&V tasks of IEEE Std.7-4.3.2 are covered in the minimum V&V tasks of IEEE Std.1012. We choose these tasks as the V&V tasks of our model. RG 1.168 mentions six tasks in the optional V&V tasks of IEEE Std.1012-1986 for the safety-critical software V&V in NPP. At the revision from the IEEE Std.101

Table 2: The V&V tasks which are used for the CB-VVCM extension.

Level	V&V tasks	Sources
5	Evaluation of user document	RG 1.168
	Regression analysis and testing	RG 1.168, IEEE Std 1228
	Test evaluation	RG 1.168, IEEE Std 1228
	Sizing and timing analysis	IEEE Std 1228
	Database analysis	IEEE Std 7-4.3.2
4	Audits	RG 1.168
	Inspection	IEEE Std 7-4.3.2
	Algorithm analysis	IEEE Std 7-4.3.2
	Control flow analysis	IEEE Std 7-4.3.2
	Data flow analysis	IEEE Std 7-4.3.2

2-1986 to the IEEE Std.1012-1998, ”Configuration management” and ”Installation and checkout testing” tasks are included in the minimum V&V tasks from the optional V&V tasks. As the result, we take four tasks from RG 1.168 for our CB-VVCM. In IEEE Std.1228, we find three V&V tasks related to the safety analysis which are overlapped the optional V&V tasks of IEEE Std.1012. Table 2 shows the result of CB-VVCM extension by adding V&V tasks for safety-critical software of NPP. The added V&V tasks are assigned to levels 4 and 5 by considering in terms of rigorousness, intesiveness, and fundamentalness. After deciding the V&V tasks which reflect the safety-critical software domain, we can construct the structure of our CB-VVCM as shown in Figure 4. It consists of the level, task, subtasks and other constituents such as input/output artifacts and implementation time. The ”Implementation time” means the specific lifecycle phase when the task is performed. The ”Related CMMI PA” presents the overlapped CMMI PAs related to the specific V&V tasks of CB-VVCM to support the parallel assessment effort with CMMI.

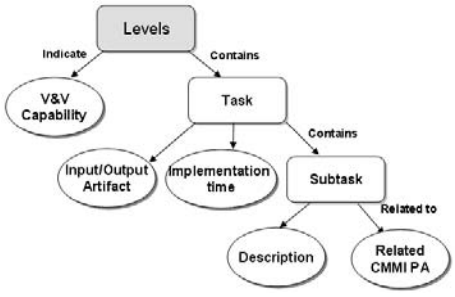


Figure 4: The structure of criticality-based V&V capability model.

Table 3: The example of comparison result : 3.1 Baseline change assessment.

CB-VVCM				CMMI		
Task	Subtask	Artifact		PA and Practice	PIIDs	
		Input	Output		Direct	Indirect
Baseline change assessment	Evaluate proposed software changes for effects on previously completed V&V tasks	·SVVP* ·Proposed changes ·Hazard analysis report ·Risks identified by V&V tasks	·Update SVVP tasks: Baseline change assessment ·Anomaly report	Configuration management SP2.1	·Change request	·Change request impact analysis ·Change request lifecycle or workflow description ·CCB/stakeholder review records ·Configuration item revision history
	Plan iteration of affected tasks or initiate new tasks to address software baseline changes or iterative development processes	·SVVP ·Proposed changes ·Hazard analysis report	·Update SVVP tasks: Baseline change assessment ·Anomaly report	Project Planning SP 2.1	·Project schedules ·Schedule dependencies ·Project budget	·Staffing profile ·Identified major mile stones ·Task dependencies
	Abbreviated					

* SVVP(Software Verification and Validation Plan

3.1.3. Analyze and extend CB-VVCM with CMMI PAs

We have two purposes in mapping CB-VVCM tasks to CMM I PAs. The first is the creation of the mapping table which can be used in conjunction with CMMI to assess both the general software process and the V&V process. The second is the derivation of additional V&V tasks from CMMI PAs to complement our model by analyzing the mapping result. Since CMMI PAs and our V&V task set differ in the format and detail-level of describing, we compare the subtasks of each our V&V task with practices of each CMMI PA. The mapping conditions are whether a subtask of our model and a practice of CMMI PA have (1)the similar purpose or contents and (2)the overlapped artifacts. To check the second condition, we compare the input/output artifacts that are described in IEEE 1012 and the direct/indirect artifacts of Practice Implementation Indicator Descriptions (PIIDs) that were used in SEI’s pilot sites [12]. For example, Table 3 presents the mapping results of ”3.1 Baseline change assessment” V&V task in level 2. Most comparision results are partial mapping with the CMMI PA viewpoint. In other words, one V&V task of our model can be mapped to no CMMI practice or partial CMMI practices of one or more PAs. It is natural because the target of CMMI is general software process and its scope is broader than that of ours.

Figure 5 describes the characteristics of the whole result using Venn diagram. In the CB-VVCM side, the non-overlapped V&V tasks are mainly the operation and maintenance related V&V tasks. In order to identify the complement point from this result, we should focus on the other side, CMMI. The non-overlapped CMMI PAs are four PAs: OID, QPM, OT, and OEI as shown in Figure 5. These four PAs are related to organizatinal process management and improvement. This result means that organizational and managerial V&V practices should be completed to make CB-VVCM more useful since CB-VVCM V&V tasks put

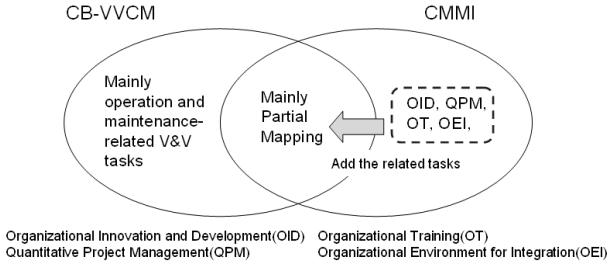


Figure 5: The characteristics of comparision result between CB-VVCM and CMMI.

much focus on technical practices. By referring the practices of OID, QPM, OT, and OEI, we derived four organizational level V&V tasks and then extended CB-VVCM like Table 4. The four tasks were assigned from level 2 to level 4 by considering its original CMMI level and the other V&V tasks in each capability level.

3.1.4. Map to the practices of ISO 9001:2000

In addition to CMMI, we mapped CB-VVCM to ISO 9001 to support to parallel assessment and find the complement point. However, it is too difficult to compare because the clauses of ISO 9001 are described conceptually and contain few V&V issues. Therefore, we performed the initial comparision between CB-VVCM and ISO 9001 with the mapping result between CB-VVCM and CMMI of the previous step and the mapping result between CMMI and ISO 9001 [13]. And then we aligned and modified the initial result by comparing again the V&V tasks of CB-VVCM and their initial mapped clauses of ISO 9001. Table 5 presents the part of comparision result of step 3 and 4 which belongs to level 4. The results of other levels are abbreviated becasue of space.

Table 4: The V&V tasks which are derived from the non-overlapped PAs of CMMI.

Level	V&V tasks	Source (level)
4	Evaluating and improving V&V process	OID (5)
	Quantitative management review of V&V	QPM (4)
3	Establishing organizational policy and commitee for guaranteeing independent V&V	OEI (3) (IEEE Std.7-4.3.2)
2	Establishing managerial and technical V&V training program	OT (3)

Table 5: The part of final mapping table among CB-VVCM, CMMI and ISO 9001 in level 4.

Level	Task	CMMI	ISO 9001
4	Interface with organizational supporting processes	OPD SP1.1, IT SP2.1, REQM SP1.2, VER SP1.3, VAL SP1.3	4.1, 4.2.1, 4.2.2, 7.2.2, 7.2.3, 7.3.5, 7.3.6, 7.5.2, 8.2.4, 8.3
	Management and technical review support	IT SP1.1, REQM SP1.5, VER SP3.1, VAL SP2.1	7.2.2, 7.3.5, 7.3.6, 7.5.2, 8.2.4, 8.3
	Quantitative management review of V&V	MA SP1.3, 1.4, 2.1, 2.2, 2.3, 2.4, QPM SP1.1, 1.4, 2.1, 2.3, P2.4	5.4.1, 7.1, 8.1, 8.2, 8.4, 8.5.1
	Evaluating and improving V&V process	OID SP1.2, 1.4, 2.1, 2.2, 2.3	8.5.1
	Hazard analysis	TS SP1.1, 2.1, 3.1, VER SP3.1, VAL SP2.1	7.3.1, 7.3.3, 7.4.1, 7.5.1, 7.3.5, 7.3.6, 7.5.2, 8.2.4, 8.3
	Configuration management assessment	CM SP3.2, VAL SP2.1, VER SP3.1	4.2.3, 4.2.4, 7.3.1, 7.3.7, 7.3.5, 7.3.6, 7.5.2, 7.5.3, 8.2.4, 8.3
	Risk analysis	RSKM SP2.1, 2.2, 3.1, DAR SP1.6, VER SP3.1, VAL SP2.1	7.3.5, 7.3.6, 7.5.2, 8.2.4
	Audits	VER SP3.1	7.3.5, 8.2.4, 8.3
	Inspection	VER SP3.1	7.3.5, 8.2.4, 8.3
	Algorithm analysis	VER SP3.1	7.3.5, 8.2.4, 8.3
	Abbreviated		

3.2. Criticality-based assessment framework of V&V capability

CB-VVCM which is core constituent of our framework and the mapping table with other models were developed according to the development steps of the previous section. CB-VVCM consists of five capability levels and 47 tasks as shown in Figure 6. In addition to Table 5, the mapping table contains the artifact information of CB-VVCM and CMMI to support the assessment at the same time. The questionnaire for the assessment which covers the contents of CB-VVCM is designed as shown in Figure 7 by referring the maturity questionnire form of CMM [14]. Finally, the ranking policy follows that of CMMI.

4. ANALYSIS

In this section, we present the analysis of our framework through the industry's practical viewpoint and the comparision with TMM.

4.1. Feedback from industry's feasibility study

The CB-VVCM was reviewed by a Korean company which provides the NPP design engineering and the plant operation and maintenance service. This company mainly devel-

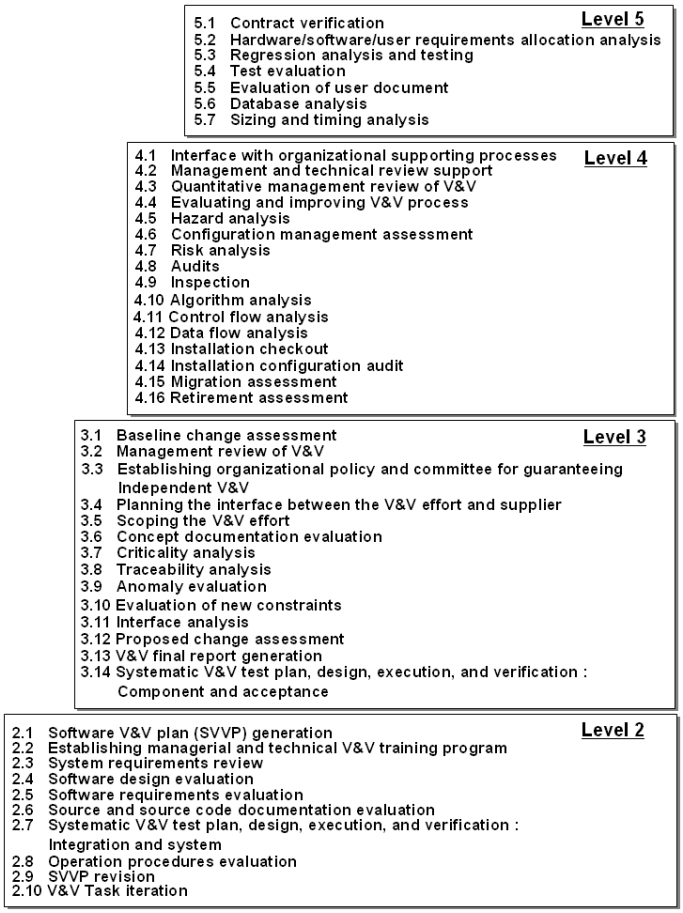


Figure 6: The five levels and tasks of CB-VVCM.

ops safety-critical systems of NPP through acquisition. Although this company's acquisition process emphasizes on the V&V, the evaluation points related to quality are usually assigned low rates in Request For Proposals(RFP). To emphasize the importance of quality and promote the efficient assessment of supplier's V&V capability during acquisition, this company plans to apply our framework to its acquisition process as a tool for assessing supplier's V&V capability. Before applying, its internal experts reviewed our framework and applied it preliminarily to two finished real projects. The one of projects doesn't have the required artifacts of our framework except project plan, source codes

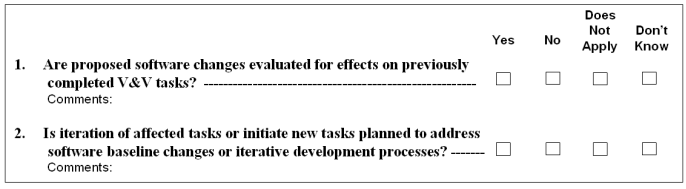


Figure 7: The questionnaire example of our framework.

and manuals. On the other hand, the other project provides 46 artifacts such as software verification report(requirement, design, code), software review report, test procedure, etc. We can easily perform the V&V capability assessment of this project's supplier through checking whether the recommended artifacts existed in the real artifacts. However, the assessment of the organizational V&V tasks was impossible because the assessment of these tasks needed the internal organizational information as evidence. Through the expert's review and the feasibility studies, we received the following feedback :

- Since the V&V tasks and subtasks consist of the specific practices in the field, they are easy to understand during the assessment.
- Since the V&V tasks provide the specific artifacts, the assessment is easy to perform through checking whether the artifact is produced or not.
- Since an artifact contain the other work products of one or more tasks or the terms used in the real artifacts always does not completely match our recommended artifacts, the detailed descriptions of contents of artifact are needed. For example, in the case of the second project, its "SOFTWARE REVIEW REPORT/REQUIREMENTS REVIEW XXX PROGRAMMABLE DIGITAL COMPARATORS" contains the result of traceability analysis, software requirement evaluation, etc. Our framework specifies that the traceability analysis task report, software requirement evaluation task report, etc. should exist to show the evidence of performing the traceability analysis and software requirement evaluation. To make the assessment more easier and exacter, the provision of detailed structure and description of artifact's contents is required in addition to the provision of the list of recommended artifacts.
- The V&V capability level and the assigned tasks are appropriate to use. However, the granularity of each V&V tasks should be adjusted.

After completing some feasibility studies in the near future, this company plans to develop a web-based assessment supporting tool to use in the acquisition process. By asking supplier candidates to answer question and upload documents, this company will execute the assessment automatically. The assessor can audit using this system and monitor the result of assessment. For the organizational V&V tasks, this company has an intention to make the policy that the assessor directly audits the supplier candidate's organizational information within its opening boundary. We recommend that the company which has V&V capability level 4 or more can be accepted because the rigorous and intensive V&V tasks are included in level 4 or more.

Table 6: The comparision our framework with CMM.

	Our framework	**TMM**
Objective	Assessment of V&V capability of organization	Assessment and improvement of testing maturity of organization
Application scope	Safety-critical software organization (internal, external)	General testing oranization (internal)
Structure	- Five capability level - 47 V&V tasks	- Five maturity level - 13 goals and 43 subgoals
Supporting parallel assessment	CMMI and ISO 9001	SW-CMM, CMMI and ISO 9001
Features	- Consisting of the spcific V&V practice focusing on safety-criticality - Providing the recommanded artifacts list to support the assessment - Compliance with the international standards	- Providing the specific assessment process - Supporting the testing process improvement activities - Reflecting the industy-wide software testing evolution

4.2. Comparision with TMM

TMM is one of our related assessment models. We compare our framework with TMM on the following criteria: objective, application scope, structure, supporting parallel assessment with other model, and features. The result is presented in Table 6. Although TMM is widely used in the assessment of testing process, its limitations with the viewpoint of assessing the criticality-based V&V capability are followings:

- TMM does not provide or emphasis on the safety-critical specific V&V practices because the target is general software testing organization.
- When the target is the external supplier whose internal information related to the organization is difficult to be open, the assessment based on TMM is hard to perform. Because the large portion of TMM covers the organizational level tasks, the organizational information is needed as evidence during assessment.
- Since the structured component of TMM such as subgoal, practices etc. does not provide the specific artifact list which are recommended, the assessment can be difficult to find proper kinds of evidences.
- Since the meaning of ”Testing” in TMM is ambiguous, it may not proper to apply V&V activies except testing activity. According to its guidebook [15], some of its practices in maturity goals mentions specific testing methods such as black/white box test, multilevel test. However, it does not explain any similiar methods as safety analysis or criticality analysis for requirement or design.

Consequently, our framework is more appropriate to use as the assessment tool of safety-critical software organization.

5. CONCLUSION AND FUTURE WORK

In this paper, we proposed the framework for V&V capability assessment focused on the safety-criticality to assist the safety-criticality V&V assessment. To provide the essential V&V practices and the confidential capability level scheme for the V&V assessment, the criticality-based software integrity level and the minimum V&V tasks of IEEE Std.1012 are used as the basis. The recommended V&V tasks of IEEE Std.7-4.3.2, 1228 and RG 1.168 are referenced to reflect the V&V which are emphasized in the safety-critical system. Through mapping our V&V tasks to the CMMI PAs and ISO 9001 clauses, we found the weak point of our framework and then added the organizational level V&V tasks to complement it. This mapping table can support to the parallel assessment with CMMI and ISO 9001. Using our framework, the safety-critical software organization which wants to assess the external supplier's V&V capablity can perform the assessment easily and efficiently since our questionnaire which is made of the specific V&V tasks and its related artifacts make the assessor easily understand its purposes and contents. Specially, we can apply our framework in the initial steps of acquisition process. In addition, we expect that our framework can be used as the criticality-based V&V process improvement model by enforcing the compliance of minimum V&V tasks according to the criticality-based V&V capability level. Consequently, through using our framework as a tool for the V&V capability assessment and/or V&V process improvement, the acquirers of safety-critical software project can select the proper supplier and expect the high-quality safety-critical software, and the internal/external suppliers can initiate the V&V process improvement activities based on the assessment result. Finally, we plan to improve our framework by reflecting the feedback of the industry such as adjusting the granuality of each V&V tasks and developing the specific description of V&V subtasks using the realistic terms after performing the several fesibility studies in the near future.

6. REFERENCES

[1] Nancy G. Leveson,“Software Safety:Why, What, and How,” *ACM Computing Surveys*, Volumn 18, Issue2, June 1986.

[2] M.B. Chrissis, M. Konrad, S. Shrum, *CMMI: Guidelines for Process Integration and Product Improvement,* Addison-Wesley, 2003.

[3] I.Burnstein, A.Homyen, T.Suwanassart, G.Saxena, R.Grom,"A testing maturity model for software test process assessment and improvement," *Software Quality Professional*(American Society for Quality), Vol.1, No.4, pp.8-21, 1999.

[4] *IEEE Std. 1012-1998 - IEEE Standard for Software Verification and Validation, IEEE Computer Society,* 1998.

[5] *IEEE Std. 7-4.3.2-1993 - IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations, IEEE Standards Board,* 1993.

[6] *IEEE Std. 7-4.3.2-2003 - IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations, IEEE Power Engineering Society,* 2003.

[7] *IEEE Std. 1228-1994 - IEEE Standard for Software Safety Plans, IEEE Standards Board,* 2003.

[8] *IEEE Std. 1012-1986 - IEEE Standard for Software Verification and Validation Plans, IEEE Computer Society,* 1986.

[9] *Regulatory Guide 1.168 - Verification, Validation, Reviews, And Audits For Digital Computer Software Used in Safety Systems of Nuclear Power Plants, U.S Nuclear Regulatory Commission Office of Nuclear Regulatory Research,* 1997.

[10] *ISO 9001:2000 - Quality management systems - Requirements, ISO* 2000.

[11] Patricia Rodriguez Dapena, *Software Safety Verification in Critical Software Intensive Systems, PhD dissertation,* Eindhoven University of Technology, 2002.

[12] *CMMI in Small Settings Toolkit Repository from AMRDEC SED Pilot Sites; DRAFT 14, http://www.sei.cmu.edu/ttp/publications/toolkit/InitialCMMIGapAnalysisUsingPIIDs.html, SEI,* 2004.

[13] Boris Mutafelija and Harvey Stromberg, "Systematic Process Improvement Using ISO 9001:2000 and CMMI," Artech House, 2003

[14] S.Masters, C.Bothwell, *Maturity questionnaire,* Technical Report, CMU/SEI-94-SR-7, SEI, 1994.

[15] Ilene Burnstein, *Practical Software Testing,* Springer, 2003.

Workshop 2.1:
Workshop on Evolution of Software Systems in a Business Context

Session:
Tools and Architectures for Evolution

Requirement Engineering and Study of Software Evolution by Using an Open Source Bug Reporting Tool

Parastoo Mohagheghi[1,2], Reidar Conradi[1,3]

[1]Department of Computer and Information Science
Norwegian University of Science and Technology, NO-7491 Trondheim, Norway,
[2]Agder University College, No- 4898 Grimstad, Norway
[3]Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway
parastoo@idi.ntnu.no, conradi@idi.ntnu.no

Abstract

This short paper reports work in progress on planning and studying evolution of a product using Trac as an open source bug reporting tool. We have extended Trac to allow collecting data on requirement and defect types, their origin, and their impact on product characteristics. The solution allows integration of all change data in a single tool and study of different aspects of software evolution. It also supports project management in requirement management, planning of releases and allocating resources.

Problem statement

Most of today's software products are in continuous evolution. Previous releases should be maintained (*corrective maintenance* or repairing faults), while new features are added to the product or its performance is improved (*perfective maintenance*), software undergoes refactoring or redesign (*perfective maintenance* to prevent faults), and is adapted to new environments (*adaptive maintenance*). Sometimes other terms are used for maintenance and evolution activities, or these terms are used differently, making comparison of studies difficult. The term *evolution* is often used to cover maintenance activities while the product is still growing. Earlier studies have tried to study the ratio between different categories of maintenance activities, the origin of changes, or the impact of changes on project schedule or defect-density (see [Mohagheghi et al. 2004]). However, empirical studies on software evolution are few, there are often serious validity concerns due to missing and incomplete data, and the researcher has had no control on data collection (thus bounding the research questions to the scarce available data).

Open Source Software (OSS) products are examples of continuously evolving software products delivered in short releases. Due to free access to data on software code and change logs, evolution of OSS products has been studied in terms of growth of software size, the number of changed or added source code modules, or change logs from CVS and similar version control systems. Among the studies are [Chen et al. 2004] [Capliuppi 2003] [Antoniol et al. 2004] [Scacchi 2004]. While these studies provide valuable information on the rate of growth of products, other questions on the origin of changes (e.g. changed environment, market needs etc.) or type of changes (extensions, perfection, functional vs. non-functional) are difficult to answer. One challenge is to integrate data from several sources such as version control systems, release notes and defect reporting

systems. Chen et al. have studied change logs of three OSS products (small and medium-sized products), and compared these to changes in the source code. Their results showed that change log files are incomplete for research purposes due to the relative high percentage of omissions. The authors have also categorized changes as being corrective, enhancement (perfective and adaptive), a code rearrangement (preventive) or a comment change. For two of the products, the majority of changes are enhancements, while for the third one corrective changes dominate. They mentioned that categorization of changes need fairly descriptive change logs and is subjective. Our study of a commercial product [Mohagheghi et al. 2004] also confirmed the need for integrating data on different types of changes to develop models on evolution.

OSS products and many commercial ones use open source bug reporting tools such as Bugzilla [Bugzilla] or Trac [Trac], where a *bug* may be a defect or an enhancement request. In both tools, the severity of a bug may vary from "blocker" to "trivial", or set to "enhancement". Using the severity field for enhancements is not an ideal solution since enhancements may also be valued as critical, highly desirable, or minor cosmetic. However, storing enhancements and defects in the same tool allows bringing different types of maintenance together and collecting data in a consistent way. Such tools allow automatic generation of reports or statistics, are integrated with relational databases and allow search for special keywords or entries.

Extending Trac for management support and study of evolution

In the SEVO project (Software EVOlution in component-based software engineering) [SEVO], we are interested in studying how and why software products evolve. One of the products being followed is a tool for analyzing accessibility of websites. The project is financed by EU and the tool will be an OSS product being developed in three releases by multiple partners across national boundaries. To address requirement engineering of the product and answer research questions on the origin of changes or their impact on product quality, we have decided to extend Trac with new fields. Trac is chosen since adding fields is easy and it is well integrated with Subversion [Subversion], which is an open source version control system selected for the product. The extensions will help project management in communicating requirements to developers, planning milestones, and allocating requirements to developers and system components. Our approach is close to *action research*; done collectively by researchers and developers in repeated cycles of data collection and improving the process of requirement engineering. The character of the project allows research in addition to development (or in fact research is development).

Certain data such as a unique identifier, reporter's name, severity, and component name are included for each report. However, additional data are required to address management needs and research questions. Among the extensions are:

- We have added the possibility to indicate whether an enhancement is mandatory, highly desirable, desirable, optional or cosmetic. This will allow prioritizing requirements based on their value for customers.

- We have added the possibility to indicate the impact of an enhancement or defect on product characteristics such as functionality, reliability, availability, performance etc. This will again help prioritizing requirements by showing the perceived impact on product characteristics. It will also allow deciding test scenarios and categorization of changes for study of evolution.
- Both defects and enhancements will have a history field. For defects, it may indicate whether the defect is in a new part of the product, is a latent defect from a previous release, is a result of redesign, or is injected after a fix. Such data helps to evaluate the quality of a release and the efficiency of verification techniques in detecting defects. For enhancements, the history will indicate whether it is a new requirement, a modified requirement, a request for redesign or adapting to a new environment. It will allow classification of evolution types and generating metrics such as requirement instability or requirement creep.

Existing fields in Trac such as “affected components”, “milestone to solve” and “assigned to” are used to evaluate the impact of defects or enhancements, plan future releases, and assign these to teams.

Benefits of the solution and conclusions

The benefits of the approach (in addition to following the status of defects and normal corrective tasks) are:

- Using a single available tool to get an *overview of all changes* made to a release in terms of number and type of changes.
- Managing requirements and defects, assigning resources and planning milestones; thus *management support* for evolution planning.
- Providing *traceability* in accordance to the “IEEE recommended practice for software requirements specifications” (IEEE std 830-1998). Each requirement will have a unique identifier, and the origin and impact of a requirement are specified. It is also desirable to ascertain the complete set of requirements that may be affected by modifications in design (or redesign).
- *Evaluating data* needed to study evolution types and requirement evolution over releases. This knowledge will be used to refine research questions and plan data collection in other projects.

We should also study how data from Subversion (such as change logs and software modules checked in for a change) can be integrated with data from Trac to study the impact of changes on source code. Probably, the project should agree on rules such as including Trac identifier in change logs and checking in modules with one change at a time. The extended Trac system has been in use for a few weeks in the first release and we will follow its use over time.

References

[Antoniol et al. 2004] Antoniol, G., Di Penta, M., Gall, H. and Pinzger, M.: Towards the Integration of Versioning Systems, Bug Reports and Source Code Meta-Models. Preliminary version on http://seal.ifi.unizh.ch/fileadmin/User_Filemount/Publications/giulio-setra04.pdf

[IEEE] IEEE Std 610.12-1990 (R2002), http://standards.ieee.org

[Bugzilla] http://www.bugzilla.org/

[Capliuppi 2003] Capiluppi, A.: Models for the Evolution of OS projects. *Proc. Int'l Conference on Software Maintenance (ICSM'03)*, pp. 65-74.

[Chen et al. 2004] Chen, K., Schach, S.R., Yu, L., Offutt, J. and Heller, G.Z.: Open Source Change Logs. *Empirical Software Engineering*, 9(2004), pp. 197-210.

[Mohagheghi et al. 2004] Mohagheghi, P. and Conradi, R: An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes. *Proc. of the ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2004)*, IEEE CS Order Number P2165, pp. 7-16.

[Trac] http://projects.edgewall.com/trac/

[Scacchi 2004] Homepage of W. Scacchi: http://www.ics.uci.edu/~wscacchi/

[SEVO] http://www.idi.ntnu.no/grupper/su/sevo/index.html

[Subversion] http://subversion.tigris.org/

Software Architecture for Evolving Environment

Jaroslav Král and Michal Žemlička
Charles University, Faculty of Mathematics and Physics
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
{jaroslav.kral,michal.zemlicka}@mff.cuni.cz

Extended abstract

1 Introduction

Companies evolve during time. They grow, they change or extend the business topic, buy or sell their part(s), cooperate with other companies, etc. For classical information systems supporting such company most of the changes above mean significant modifications or even full replacement by newer system.

Such changes are very risky: the switching to new system may take long time during that the company is without its supporting information system, the new system may have new bugs, the new system typically require significant retraining of its users what causes (except significant expenses) risk that they will make new errors and that the users – at least for some time – would not be as effective as before.

We therefore need to keep the system as stable as possible what could reduce above mentioned risks, software costs, and user discontentment. Software architecture proposed in this paper can significantly reduce these risks and costs.

2 Software Confederation

2.1 Idea

In human society there are complex teams performing complex services. These services are available through very simple interfaces that tend to be very stable in time – even in the case when the implementation of the service significantly changes.

This feature is significant for the service users that do not need to know about the changes in the service implementation and also the service users do not need to change their access to the service. They simply use it.

In the company there are also several services like accounting, manufacturing, shipping, etc. Such services have natural language interfaces – sometimes even several ones: for each group of users one. These interfaces are as general as possible and as precise as necessary (they are usually declarative), they tend to be very stable – sometimes even decades or more.

People also tend to be lazy and therefore interfaces of real-world services are very compact (effective).

We expect (and practice is proving our expectations) that the use of software equivalents of interfaces of real-world services brings stability and effectiveness to the interfaces of software applications or components providing functionality of (or supporting) the real-world services.

2.2 Structure

Software confederation is a virtual peer-to-peer network of permanently available applications providing some functionality. Such network is equipped with global interface for general users and with local interfaces of the individual services available for the expert users responsible for the proper work of the services.

We distinguish these service/component types:

- Application components,
- front-end gates,
- infrastructure components,
- portals,
- data stores,
- process managers.

We expect that there will be more components of the same type in a confederation.

2.2.1 Application components.

Application components (AC) are the application providing the real functionality of the confederation. ACs are typically specialized applications providing some functionality (e.g. bookkeeping). ACs have their own user interface and are used directly by the users being experts in given problem domain. ACs can be legacy systems, third party systems, or newly developed applications.

Application services are connected to the rest of the software confederation by a special interface (*primary gate*) that wrap the application so that its interface is converted to sending and receiving messages.

It can be expected that interface of such application is procedural and implementation-oriented what makes such interface unusable for direct use by the rest of the confederation (also for the security and safety reasons).

Therefore the application components are equipped by front-end gates providing the high-level interfaces to the ACs. It is common that an application component is equipped by several front-end gates supporting interface for different groups of users (people or other applications). It is possible to treat logically one application component with all its front-end gates as a single component.

2.2.2 Front-end gates.

Front-end gates (FEG) are specialized applications that convert application component's procedural interface to declarative interface available to other components of the confederation. FEG play the several roles:

- of interface convertor transforming required high level declarative and problem-oriented interface to the available procedural and application-oriented interface provided by the primary gate of the application component.
- of interface customization – every group of users can have their own FEG providing interface to the application component exactly matching the rights and needs of given user group.
- of change stopper – it is very likely that implementation changes in the application component would not affect the high-level interfaces provided by the front-end gates. It allows keeping the changes local and manageable.

2.2.3 Infrastructure components.

Infrastructure components are responsible for redirection, advanced addressing and many other useful things within confederation. From the logical point of view they can play e.g. the role of post offices.

2.2.4 Portals.

Portals provide interface to the confederation and its services "for the rest of the world". There can be more portals in one confederation as there can be more different groups of users. For example, there can be one portal for the company employees and one for company customers.

2.2.5 Data stores.

In real-world services is the task scheduling more complex than "first in, first out (served)". Usually the tasks are scheduled according some complex criteria and

provided according this schedule. By the software services it can be similar: the request can be collected and scheduled.

Usage of data stores (or, more precisely, services working like data stores) can be useful also in other sense: When data for processing are taken from one data store and the results are stored in other data store, the batch processing can be used. The use of batch processing simplifies integration of some legacy applications and also design and implementation of newly developed batch working applications.

2.2.6 Process managers.

Many tasks consist of several subtasks (steps). Order of these subtasks may be fixed, but sometimes it can vary. Performance of complex tasks can be driven by workflows, Petri nets or other structures describing how the task should be solved.

The business processes (BP) are driven by such process managers. BP can be described by different methodologies. Different people in a company may be more convenient with different process description methodologies. There can be also different types of business processes, what can also conclude that more different BP definition methodologies should be available. Hence there should be often more process managers supporting different process description methodologies.

The concrete use of individual process manager can be given by the supported task or by the person who is responsible for the task.

Presence of process managers in the system is crucial: it gives an opportunity to define or change business processes after the development of the system. It also changes the system analysis, design, and development: it is not necessary to describe business processes in the analytical phase of the development; it is enough to find out the necessary substeps and built support for them. The final business processes must be set, controlled and configured by the people responsible for them.

2.3 Features

Software confederations are one of the possible models of service orientation. They are expected to be used especially for large and complex (information) systems.

Software confederations can be developed incrementally, they can integrate legacy systems, and they can evolve in time. Many changes can be made inside single application component or by modifying business processes. Hence from the software developers and for end-users they behave very stable.

Software confederations support modern managerial turns as in the examples below:

Acquisition of a new division – if the information systems of both the division and the enterprise are of the confederative structure, the integration is quite easy, as corresponding services would have very similar interfaces. The integration then could be done by changes in some front-end gates and by some changes in portals.

When only the enterprise has its information system in the form of software confederation, the information system of the new division can be wrapped and equipped by front end gates and then used (integrated) as a new application component.

Joining of two or more enterprises – the situation is similar as in the previous case. When there are more application services providing the same or very similar their functionality, request for their services can be redirected by infrastructure services according rules set by new management.

Outsourcing – as the individual organizational substructures are supported by individual application services (or at least by individual copies of application services) it is quite easy to separate such service and reconnect it to the system to some special security gates. (The ousourcing provider need not know too much about the enterprise.)

A very good feature is that end users of many application services do not encounter any (or quite any) changes of their applications.

3 Conclusion

A software architecture supporting smooth evolution of an information system has been proposed. It has been shown that many managerial turns that cause significant rewriting of the typical enterprise information systems, can be supported in software confederation using local changes only.

Workshop 2.1:
Workshop on Evolution of Software Systems in a Business Context

Session:
Reverse Engineering of Business Rules

A Framework for Exacting Workflows from E-Commerce Systems

Maokeng Hung[1] and Ying Zou[2]
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, K7L 3N6, Canada
alex.hung@ece.queensu.ca[1], ying.zou@queensu.ca[2]

Abstract

For many organizations, reacting to fast changes in customers' requirements becomes keys to maintain the competitive edge in the market. However, developers must locate business logics manually in source code in order to add new requirements. This problem has caused maintenance costs to escalate while budgets and the corporate spending are shrinking. In this paper, we propose an automatic approach that extracts workflows from sources code and utilizes control structure information in as-specified workflows to refine the recovered as-implemented workflows from source code. By using the as-specified workflows to guide our extraction, we can limit the searching scope for business logics and locate explicit associations between artifacts in the as-specified and as-implemented workflows.

1. Introduction

Business logics can be considered as requirements and conditions on how to manipulate data in information systems [3]. As an example of business logic, a bookstore may have different rules to calculate the price of books sold based on a variety of coupons and special sales conditions. A business process communicates the knowledge of organizational structures, business policies, and business operations. For instance, when a book is ordered, the business process consists of checking the availability of the books, restocking the inventory if needed, and validating the buyer's credit card. Typically, a workflow is a computerized representation of a business process that consists of a sequence of tasks that implement business logics (rules), control/data flows that link through tasks, participants, and resources required by tasks.

In order to achieve dramatic improvements in critical measures of performance, such as cost, quality, and speed [7], organizations change continuously and constantly to reflect the rethinking and redesign of business logics and organizational structures in business processes. Information Technology (IT) departments are faced with the growing challenges of supporting frequent requests for changing business logics. For many IT departments, the challenge is compounded by complex architectures and distributed computing infrastructures that were developed over the last 10-20 years. In particular, business logics are often hard-coded in such systems. The documentation of workflows is often lost, obsolete or never existed. Therefore, it is a challenging job for developers to manually locate the code blocks that implement business logics, and to modify the code to meet new requirements. This situation has caused system maintenance costs to escalate while budgets and corporate spending are shrinking.

In this context, a number of research has been carried out to recover business processes from source code [2…6]. However, most of the proposed approaches focus on extracting business logics without generating the workflows (the underling structure) implemented by applications. Consequently, the ***as-specified workflows*** (i.e., documented business processes) cannot be updated. Moreover, these approaches either analyze the code syntactically or require significant human intervention to identify the semantic meanings of business logics during the extraction process [6]. In our previous work, we proposed an approach to identify business logics from the source code based on code heuristics and to extract ***as-implemented workflows*** from the source code using Abstract Syntax Trees (ASTs) [1]. Unfortunately, the extracted business logics vary in the level of granularity. For example, some identified business logics may need to be further decomposed into multiple business logics, while others may require to be merged.

The objective of this paper is to improve the precision of the workflow extraction and to reduce the requirement of the human intervention by utilizing the design information from the as-specified workflows, which serve as guidance to refine the extracted as-implemented workflows. A behavioral model is proposed to compare the structural and functional behaviors of these types of workflows (i.e., as-specified workflows and as-implemented workflows). The similarities in structures and behaviors of both types of workflows can indicate relevant code segments where business logics may be missing or over-identified.

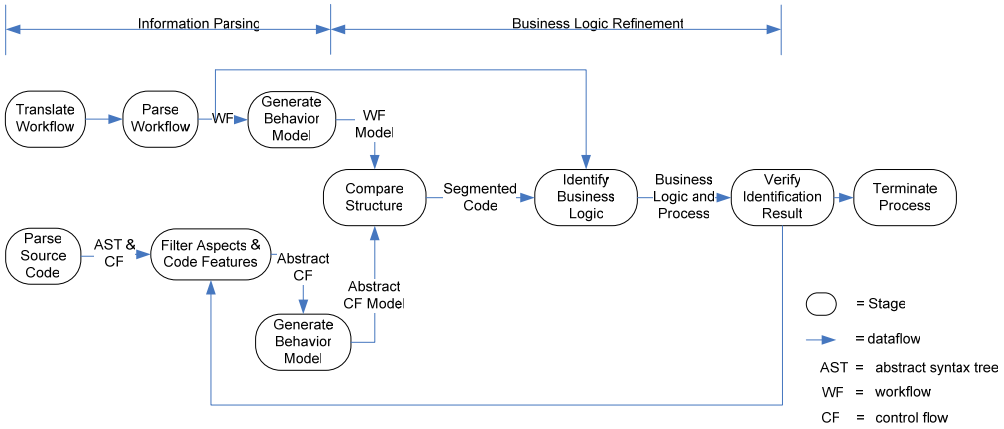


Figure 1 - The recovery consists of two major stages, information parsing and business logic refinement

2. Approach for Structure Guided Workflow Extraction

2.1 A Framework for Workflow Extraction

In most e-commerce systems, the workflows are implemented and executed without interacting with workflow engines and involving few human tasks. In such systems, software developers implement the workflow processes and their sub-processes in one system [1, 10]. Specifically, tasks are implemented as classes and methods. The business rules are hard-coded as the control and data flows in the source code. When updates are required, developers have to inspect the source code and determine the locations where changes must be made; it is our objective of this research to extract the workflows and business logics from the source code of the e-commerce systems to facilitate code updates for changing business requirements

Conceptually, workflows convey the execution of an application. An as-specified workflow can be considered as a high-level description of the control flow inside an application. On the other hand, an as-implemented workflow is extracted from the low-level implementation that realizes the as-specified workflow. Therefore, we can leverage the control structures of as-specified workflows to derive more precise as-implemented workflows at the appropriate abstraction. We proposed an approach that uses structural information for extracting workflows. The overall steps are illustrated in Figure 1.

In information parsing, we gather structural information from two artifacts – the as-specified workflows and the source code, as illustrated in Figure 1. First we generate the as-implemented workflows from the source code. Second, we raise the abstraction level of the generated as-implemented workflows by filtering programming specific features that are not in the business domain (e.g., checking object initialization). Third, we translate the workflows described by workflow description languages to as-specified workflows. Last, we convert both of the as-implemented workflows and as-specified workflows to an abstract behavioral model, which represents the control flow artifacts at an intermediate abstraction level and bridges the gap of the difference between the two artifacts.

In the stage of business logic refinement, we compare the behavioral models from as-specified and as-implemented workflows. We consider control constructs that affect the execution path of a workflow as critical nodes .e.g., Loop, Choice and Parallel). We collect a set of semantic equivalent critical nodes, compare their structural similarities, and establish the synchronization between the as-specified workflow and the as-implemented workflow in terms of segments.

To this extent, we can refine the extracted business logics within the scope of each synchronized segments. We can decompose coarse-grained business logics, or encapsulate a set of fine-grained business logics. Moreover, the precision of the extraction can be enhanced by the assistance of the data information collected from both types of workflows. In particular, we utilize the input and output data information in the as-specified workflows to determine the boundaries between business logics in source code

2.2 As-Implemented Workflow Extraction

Generating a complete trace record would produce a workflow graph that contains a large number of entities that are outside our interests. As a result, we adapted static tracing to identify business logics from source code using a set of heuristics to prune the number of code entities in the trace records. As discussed in [1], the heuristics include: 1) Utility code, 2) Java objects 3) Exception objects that should be filtered and 4) Data objects, 5) Task objects 6) User-defined class objects that are part of the business logics.

While code-heuristic recovery can be effective, various programming styles and the use of the encapsulation principle in software design has limited its usefulness to particular domains, and created challenges to extract business logics in an appropriate level of granularity. Therefore, we also apply the software metrics and documentary structure of the source code.

As discussed in [8] and [9], the documentary structures, such as the comments and white spaces, are arranged by programmers to differentiate the distance between different concepts. In practice, developers focus on a single task at a time and group the related statements into a code segment. They insert comments and white spaces between each function, (i.e. business logic). As the result, comments and white spaces are at the positions that separate different functions to help the developers understand and locate the code segments of their interests; especially the comments that occupied one or more lines in the source code, i.e. the comments that are not inside or after a statement. Therefore, documentary structures can be considered as candidates to separate the business logics in sequences. To refine a segment that represents on business logic, we further measure the properties of the separated segments using the software metrics, such as line of code in the segment, number of input and output, cohesion and coupling metrics, and name similarity. Since business logic is a function that computes the output based on the input and conditions, the statements that belong to the same business logic should be highly dependent on each other. These metrics check the interdependency between the adjacent segments. According to the outcomes of the metric measurements, the segments can be merged into a single logic if the segments are highly interdependent; else they will be left as it-is and treated as separate logics.

2.3 Intermediate Behavior Model for As-Specified and As-Implemented Workflow

An as-specified workflow has a higher level of abstraction than an as-implemented workflow. it is necessary to bring the both types of workflows to a common ground in order to perform the comparison between the two. In our research, we have developed an abstract behavioral model to represent control structures and their execution behaviors of as-specified and as-implemented workflows in a unified form.

As shown in Figure 2, there are three types of entities, namely *Control, Task and Data* to capture the control flow, the business logic or process, and input and output in the workflows, respectively. Each type contains the concrete entities for the low level behaviors in the workflows; for instance, *ControlEntity* contains *Choice* that models the alternatives in as-specified workflow and if-else statements in source code, and *DataEntity* can be either *Input* or *Output* of a task or a (sub)proess, as depicted in Figure 2. We aim to identify the concrete entities that represent the direct realization (e.g.,. a *Logic* is a task in the as-specified workflows) or indirect realization (e.g.,. a *Parallel* is implemented by threads in Java) in either as-implemented or as-specified workflows. This abstract behavioral model allows us to effectively compare both types of workflows.

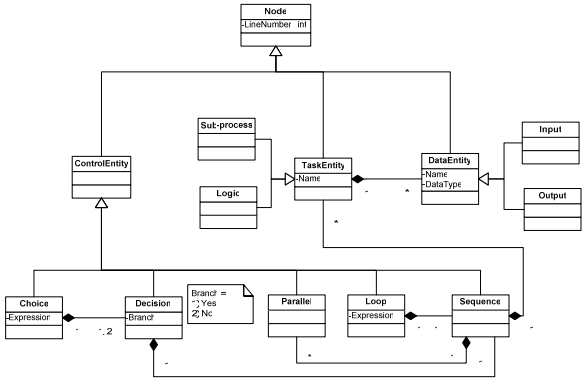


Figure 2 – Intermediate Abstraction Behavior Model

2.4 Comparison Algorithm

In our research, we developed an algorithm to compare the as-specified workflows with as-implemented workflows. Specially, the comparison algorithm attempts to match control nodes in the as-implemented workflows to the control nodes in the as-specified workflows into pairs of the matched critical nodes. The algorithm walks through both types of workflows (presented in intermediate model) until it matches the candidate critical nodes with the highest similarity. We measure the similarity of the critical nodes by the following properties:

1) Type: Are they the same type, ex. Choice nodes? The matched nodes must be the same type.

2) Order: Are they following or followed by the same types of control nodes? The nodes are more similar if their neighbor nodes are also related.

3) Depth: Does a critical node in the as-implemented workflow is equal or deeper in the nested scopes than the ones in the as-specified workflow? Since the abstraction level of the as-implemented workflows are lower than the level in as-specified workflows, the as-implemented workflows should contain more details and thus more nested scopes.

4) Inner contents: Do they have the same types of control or task nodes as children nodes? Two nodes are more similar if they both contain the same types of the control or task nodes

The pairs of the matched critical nodes serve as the boundaries of the business logics. Inside the matched

critical nodes, we map task nodes in as-implemented workflows to the task nodes in as-specified workflows, based on names, order and input and output of the tasks. Some business logics in the as-implemented workflows can be merged to increase the abstraction level. In some cases, business logics, which were not recognized in the requirement or newly added to the source code, can be identified from the comparison algorithm because no corresponding ones will be found in as-specified workflows.

3. Conclusions and Future Work

Currently, most research has focused on single source, i.e. source code, for extracting workflows from source code. In this paper we propose an approach to perform the workflow extraction by incorporating as-specified workflows modeled by business analysts. By extracting control constructs from the two artifacts and modeling them in a unified manner, we are able to measure the similarity between the as-specified workflows and as-implemented workflows. Based on this measurement, we are able to validate the conformance of the extracted as-implemented workflows, refine the abstraction without human intervention, and generate the updated as-specified workflow documents.

In the future, we plan to generalize the framework and to test it on systems in different domains. We are also interested in developers' mental models during the development and other software metrics that computes the similarity between code segments. These models and metrics will help us separate and merge the business logics from the source code with better accuracy.

References

[1] Y. Zou, T. C Lau, et al, "Model-Driven Business Process Recovery", in proceedings of the 11th Working Conference on Reverse Engineering, 2004

[2] H. Huang, et al, "Business Rule Extraction from Legacy Code", in proceedings of 20th Conference on Computer Software and Applications", 1996

[3] H. Sneed and K. Erdos, "Extracting Business logics from Source code", in proceedings of 4th International Workshop on Program Comprehension, 1996

[4] H. Sneed, "Extracting Business Logic from existing COBOL programs as a basis for Redevelopment", in proceedings of 9th International Workshop on Program Comprehension, 2001

[5] A. B. Earls, S. M. Embury and N. H. Turner, "A Method for the Manual Extraction of Business logics from Legacy Source Code", BT Technology Journal, Volume 20, 2002

[6] J. Shao, C. J. Pound, "Extracting Business Rule from Information System", BT Technol Journal, Vol 17 No 4, 1999

[7] A. Nigam and N.S. Caswell, "Business Artifacts: An Approach to Operational Specification", IBM Systems Journal, Vol. 42, No. 3, 2003.

[8] M. Folwer, "Refactoring: Improving the Design of Existing Progrmas", Addison-Wesley, 1999

[9] M. Van De Vanter, "The Documentary Structure of Source Code", to appear in Information & Software Technology, 2002

[10] D. A. Manolescu and R. E. Johnson, "A Micro Workflow Framework for Compositional Object-Oriented Software Development", in proceedings of the OOPSLA'99 Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems, 1999

Business Rules Based Web Services Oriented Customer Relationship Management System (CRM) Evolution

Yang Xu, Qing Duan and Hongji Yang
Software Technology Research Laboratory, De Montfort University, UK
{yangxu, qduan, hyang}@dmu.ac.uk

Abstract

Customer Relationship Management (CRM) is designed for helping people manage individual customer and build fixed relations with customers by responding rapidly and serving efficiently. The evolution of CRM towards Web Services will improve the flexibility of CRM *and make CRM have more efficient on data exchanging [1]. In this paper, we concentrate on the stage of extracting services from specific components based on business rules.*

1. Introduction

The failure implement cases of Customer Relationship Management (CRM) revealed its drawback on disposal business issues such as customer data integration, demand chain partner integration and more seamless access to internal and external systems [2]. The solution is making the CRM system evolve towards Web Services. Web Services could be written to expose the functionalities of any application, or to access the data of an application, even for simple data exchange through Internet [3]. Based on open standards, web services technology allows any piece of software to communicate with each other in a standardised messaging system. For Web Services can be applied as a wrapping technology around existing legacy CRM systems, new solutions can be deployed quickly and recomposed to address the changing business conditions [4]. With minimal programming, Web Services technology enables developers to easily and rapidly wrap the legacy enterprise applications and expose their functionality through a uniform and widely accessible interface over the Internet. [5]

Business rules are always changeful during organism's development. But on a time point, we assume that the extracted business rules are changeless, while the legacy CRM system evolves towards web services. Based on the stable business rules (business logic), we propose the CRM system evolution approach in following sections.

2. Approach Overview

Web-service-oriented CRM system evolution would be a complex process. In our approach, it takes the following stages:

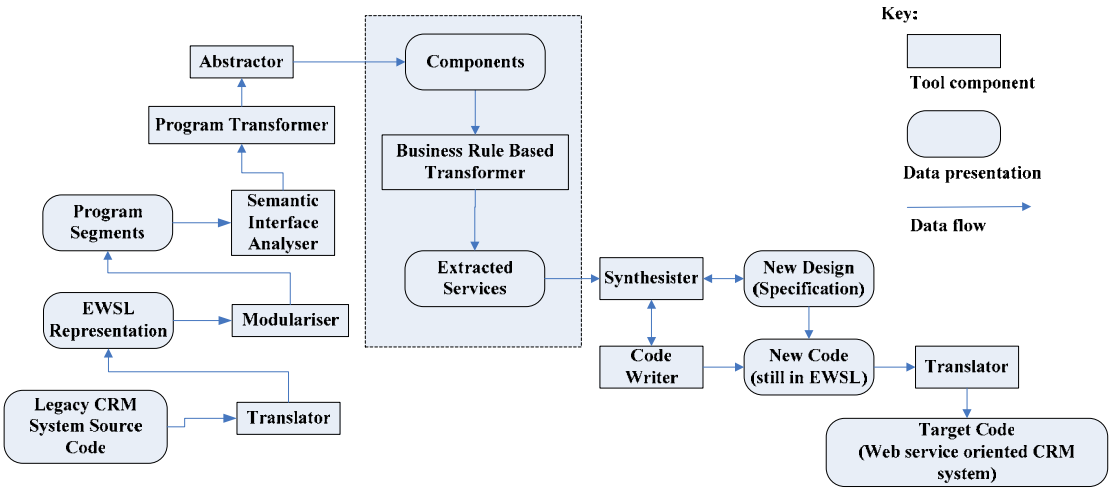


Figure 1. Process for CRM System Evolution

1) Translating source code into (Extended Wide Spectrum Language) EWSL by a language specific translator.
2) Abstracting high level specifications via abstractor from EWSL. This has been done based on the previous works [7].
3) Understanding and identifying architectural components. This stage is accomplished based upon CRM business rules (business logic), then sort out components need to be transferred.
4) Specifying services mapping with synthesised components.
5) Retargeting the web-service-oriented CRM system with implementation techniques such as wrapper, adapter and interface would be help.

In next section, Step 3 will be discussed in detail.

3. Service-Oriented Components Transformation Description

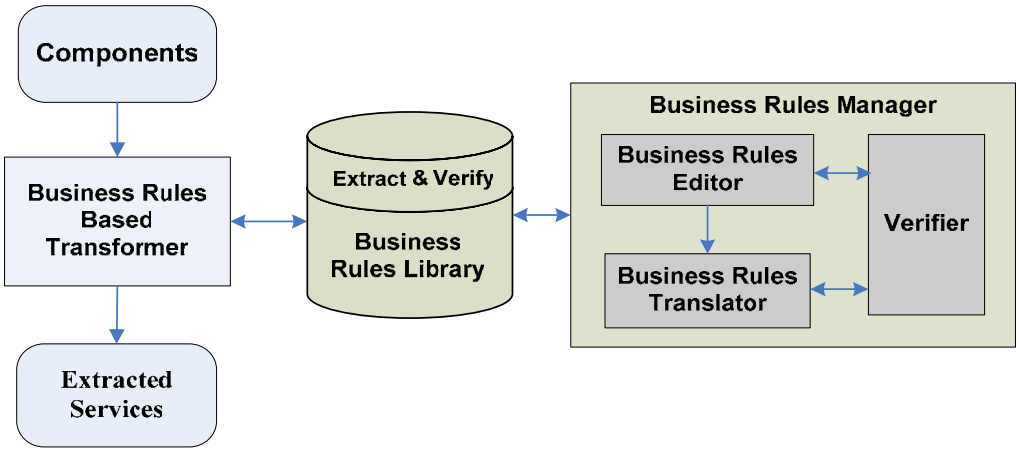


Figure 2. Business Rules Based Evolution

To achieve services mapping from synthesised components, Business Rules Manager (BRM) plays an important role. Business rules are the drivers for specifying and configuring a process. BRM captures and segregate policy decisions from the functional processes. It asserts the necessary data structures, and drives the selection of components and classes [8]. Considering the frequently changed business rules and their difference in different environments, it is efficient to define such rules using rule editor and translator and integrate them into rule management applications. When rules change, the management application allows modification of rules using a simple rule language and integrates the

changes with the application modules [9].

Sometimes hardcode parameters cannot be used to call a service because they can quickly change based on business rules. Business rules can change from time-to-time to accommodate changes in the business world. Such changes lead to changes in the software application that caters to the business function [10]. A CRM system has various modules such as workflow, client-server, database, transaction management, etc., and if business rules are hard-coded in the application, each change becomes expensive and time consuming, leading to severe maintenance problems. The separation of a CRM system model into policy, process, and structural views provides greater flexibility and more understandable representations.

1. After separating out the business logic from the CRM components, Transformer has to replace the function calls (i.e. calls to functions executing the business logic) with the call to BRM. The data is seamlessly shared with the components and new rules can be added or existing rules can be modified/deleted using the Business Rules Editor.
2. Business Rules Translator or interpreter mode of operation. Translates rules defined in the rules library into the corresponding language of the application, i.e. EWSL for various abstraction level. Rules can be expressed in simple IF_THEN_ELSE form.
3. Verifier can be invoked through a simple function/method call from anywhere in the BRM. Once invoked, verifier evaluates the relevant rules and returns the control to the application. Verifier compile rule base ensures no additional runtime overheads and provides guaranteed performance.
4. The approved rules can then be saved in a Business Rules Library. If there have some legacy rules stored in database, BRM can be used to transform them into the format instantly. In addition, deletion and updating of rules can be done in the database [11]. Once this is done, the business rules are embedded into web Services based components by BMR. Then Business Rules Based Transformer generate the code that can be compiled and linked with the modules of the application to get the final, integrated application executable file or can be compiled into EJB JAR, COM/DCOM, DLL etc.

4. An Example: Code Segment - Make CRM Components Work as Web Services

Assume a customer is new, and this customer has to introduce some personal information. In the case that the store operation would be done by a web service, the configuration for that service could be present in a database. In below Java applet code, the "CustomerStoreService" class is used to store the customer data in the database of CRM system; it is a subclass of "AbstractWebService" - a common base class for Web Services. When the client side requires a "call()" method to the Web Services, it calls the Transformer and requests the loading of response from certain components for such service; this configuration data, often cached in Business Rules Library, is used by the Transformer to create the parameter and values lists and passes them to the CRM components. While Transformer obtain the configuration required data from relative components, it passes those data to the service, calling the generic "setParameter()" method:

```
public class CustomerStoreService extends AbstractWebService {
      public void call() {
      BusinessRulesManager manager =
         new BusinessRulesManager( "CustomerStoreService" );
         //add parameters and values
      manager.prepare( this, session );
            //calls the channel
      super.call();
   }
}
public class BusinessRulesManager {
   String serviceName;
   Map cache;
   public BusinessRulesManager( String serviceName ) {
      this.serviceName = serviceName;
      initDAO();
   }
   public void prepare( WebServices service, ApplicationSession session ) {
      //loads the service configuration
      ConfigurationData conf = loadConfiguration();
      //obtain the values of the parameters, based on data session
      Map values = loadValues( session );
      //values the service parameters
      prepareService( service, conf, values );
   }
   //load cache-enabled configuration data
   ConfigurationData loadConfiguration() {
      ...
}
   //obtains data from the relative components
   Map loadValues( ApplicationSession session ) {
      ...
   }
   //sets the parameter calling the setParameter() method on the service
   prepareService( WebServices, ConfigurationData conf, Map values ) {
      ...
   }
```

5. Summary

In this paper, we describe a business rules based web services oriented CRM system evolution framework. In the framework, legacy CRM components are transferred to Web Services over the BRM. The BRM showed that the basic CRM components can be represent as Web Services by business rules editing and verifying. Our next step is working towards to create full-fledged CRM components evolution processes and utilise UML diagrams to describe.

6. Reference and Bibliography

[1] M. Davids, “How to Avoid the 10 Biggest Mistakes in CRM”, *J. Business Strategy*, Nov./Dec. 1999,

pp. 22–26.

[2] C. A. Ptak and E. Scharagenheim, “ERP: Tools, Techniques, and Applications for Integrating the Supply Chain,” CRC Press-St. Lucie Press, 1999.

[3] T. M. Chester, "Cross-Platform Integration with XML and SOAP ", *In IT Professiona l, volume 3, issue 5*, 2001, pp. 26-34.

[4] C. K. Nam and J. J. Bae, “A Framework for Processing Active Documents,” *6th International Symposium on science and technology*, 2002, pp.122-125.

[5] M. Bigatti, “Web Services Integration Patterns,” in *MySQL Users Conference, 2005,*

[6] G. Alonso and F. Casati “Web Services and Service-Oriented Architectures,” in *Proceedings of the 21st International Conference on Data Engineering*, 2005.

[7] H. Yang and M. Ward, *Successful Evolution of Software Systems.* London: Artech House, 2003.

[8] J. J Jeng, D. Flaxer and S. Kapoor, “RuleBAM: A Rule-Based Framework for Business Activity Management,” in *Proceedings of the 2004 IEEE International Conference on Services Computing (SCC’04),* 2004.

[9] L. Lin, S. Embury and B. Warboys “Business rule evolution and measures of business rule evolution,” in *Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE’03)*, 2003.

[10] A. Perkins, “Business Rules = Meta-Data,” *34th Technology of Object-Oriented Languages and Systems Conference*, 2000, pp. 285-294.

[11] A. Kovaeif and A. Groznik, “The Business Rule-Transformation Approach,” *26th International Conference information Technology Interfaces ITI*, 2004, pp.114-117.

[12] T. Puschmann and R. Alt, “Enterprise Application Integration - The Case of the Robert Bosch Group,” in *Proceedings of the 34th Hawaii International Conference on System Sciences,* 2001.

[13] L. O'Brien and D. Smith, “Working Session: Program Comprehension Strategies for Web Service and Service-oriented Architectures,” in *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, 2004.

[14] A. Groznik and A. Kovacic, “Business Renovation: From Business Process Modelling to Information System Modelling,” *24th International Conference Information Technology Interfaces IT1*, 2002, pp.405-409.

[15] I. Ho, Z. Komiya, B. Pham, H. Kobayashi and K. Yana, “An Efficient Framework for Business Software Development,” in *Proceedings of the 2003 International Conference on Cyberworlds (CW’03),* 2003.

[16] M. Wermelinger, G. Koutsoukos and J. L. Fiadeiro, “Using Coordination Contracts for Flexible Adaptation to Changing Business Rules,” in *Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE’03),* 2002.

[17] D. Sun, K. Wong and D. Moise, “Lessons Learned inWeb Site Architectures for Public Utilities,” in *Proceedings of the Fifth IEEE International Workshop on Web Site Evolution (WSE’03),* 2003.

[18] J. J. Jeng, H. Chang and J. Y. Chung, “A Policy Framework for Business Activity Management,” in *Proceedings of the IEEE International Conference on E-Commerce (CEC’03),* 2003.

UML Diagrams for the Success of Business Process Reengineering Projects

Leila Jamel Menzli[1], Sonia Ayachi Ghannouchi [2], Henda Hadjami Ben Ghézala[3]

[1] RIADI Laboratory, ENSI (National School Of Computer Science),
Campus Universitaire- La Manouba- 2010, Tunisia
leila.jamel@riadi.rnu.tn

[2] RIADI Laboratory, ISGS (Institut Supérieur de Gestion de Sousse),
B.P. 763, 4000 Sousse, Tunisia
sonia.ayachi@isgs.rnu.tn

[3] RIADI Laboratory, ENSI (National School Of Computer Science),
Campus Universitaire- La Manouba- 2010, Tunisia
henda.BG@cck.rnu.tn

Abstract. Business Process Reengineering (BPR) is a discipline in which extensive research has been carried out and numerous methodologies are emerging. But what seems to be lacking is a structured approach which supports its process. So, in this paper we develop a method to model the process supporting a BPR operation. In fact, we choose a common BPR methodology referenced in the literature and we model the process supporting it with UML diagrams. More attention is given to the second and the third steps of the reengineering process. This paper presents a structural approach, based upon UML diagrams, to fulfil the step of mapping; As-is; existing business processes in the company and the step of designin; To-be; business processes. This modelling will allow a better comprehension of BPR processes in order to improve such processes in further BPR projects.

Keywords: business process reengineering (BPR), business process (BP), business process modelling, UML use case diagram, UML activity diagram.

1. Introduction

In today s ever-changing world, the three Cs: Customer, Competition and Change itself are forcing companies to continuously improve and innovate in terms of speed, flexibility, quality, service, cost and so on. To reach these improvements, companies are on the lookout for new solutions for their business problems. Since 1990, one of the more successful business initiatives in the world is: *Business Process Reengineering (BPR)*.

Studies of BPR projects have reported very large failure rates for various reasons, but most due to the process of reengineering itself. In fact, the process of

implementing BPR is often incomplete and fragmented. The solution to this problem might be achieved in a structured approach for modelling this process.

So in this paper, we give an UML based-modelling approach for the process supporting BPR operation. This paper is organized as follows. In se ction 2 we present the concept of BPR, and in section 3 we present a survey of the state of the art of the BPR methodologies with a particularly focus on a reengineering process of a consolidated BPR methodology referenced in the literature. Section 4 presents a background information on UML modelling diagrams constructs. In section 5 we describe the UML based modelling approach for the reengineering process in BPR projects. Section 6 draws our conclusions and discussions.

2. Business Process Reengineering

Business Process Reengineering (BPR) is a popular term since the 1990s, especially after Hammer, Champy and Davenport published books to elaborate BPR related issues and cases. In fact, it was defined as the fundamental rethinking and radical redesign of business process to achieve dramatic improvements in critical, contemporary measures of performance such as cost, quality, service and speed [1].

Moreover, in the 21st century BPR has assumed a relevant role in most private and public organizations to improve existing work activities [2]. In fact, due to the evolution of organizational requirements as well as to the availability of advanced information technology, more and more organizations promote activities for restructuring and innovating their business processes (BPs). Main objectives are related to the enhancement of service and product quality and customer satisfaction. According to the reengineering paradigm, the primary focus is on process reengineering activities that require a deep understanding of organization processes and related data, to obtain a comprehensive vision of the system and, then, evaluate possible restructuring interventions [3]. So, the topic of BPR involves discovering how business processes currently operate, how to redesign these processes to eliminate the wasted redundant effort and improve efficiency, and how to implement the process changes in order to gain competitiveness.

Hence, according to these definitions and to many in the BPR field, reengineering should focus on processes and not be limited to thinking about the organizations [4]. So, one of the key concepts of BPR is *business process (BP)* [5].

So, *what is Business Process?* Definitions of business process are usually short and succinct. In [6] a clear definition was not given, but a few key features are given, such as: it contains purposeful activity, it is carried out collaboratively by a group, it often crosses functional boundaries, it is invariably driven by outside agents or customers. Jacobson [7] on the other hand describes a business process as: the set of internal activities performed to serve a customer .

Nevertheless, Bider [8] suggests to follow the most general definition given by Hammer and Champy: a business process (BP) is a series of steps designed to produce a product or a service. It includes all the activities that deliver particular results for a given customer (external or internal) [1] .

Focusing on BP, BPR is regarded as a process-oriented which is trying to overcome some problems (delays, errors and inefficiencies) raised by the classical vertical representation of the organizational structure that partitioned the firm into functional areas and departments. To support BPR project and fulfil the improvement goals, several methodologies are emerging in the literature but all balancing between changing radically existing BPs and redesigning them.

3. BPR Methodologies- A Survey of the State of The Art

The very important question while launching a BPR project in a company is: *how to reengineer?*. Several BPR methodologies are developed over the last 20 years from a variety of disciplines. But studies of BPR projects have reported very large failure rates : as high as 70% [9]. Various reasons have been given, most related to the mismanagement of the projects. In fact, the process of implementing BPR projects is often incomplete and fragmented. Some solutions to this problem were proposed based upon the proposition of structured methods supporting BPR projects [4]. In fact, to put BPR into action, Wastell and al.[10] proposed a PADM methodology (process analysis and design methodology) having four phases: process definition, baseline process selection and representation, process evaluation, and target process design. Davenport and Short [11] presented a methodology containing five steps: develop business vision and process objectives, identify redesign BPs, understand existing BPs, identify IT levers and build a prototype of the BPs.

Seven steps BPR methodologies are proposed in [12] and [13]. Similarly [14] and [15] proposed five steps methodologies turning around the Kettinger methodology [16] and which can be summarized into:

- create the vision: fix the BPR objectives,
- initialise: launch the BPR project,
- diagnose: analyse and measure the existing BPs, the allocated resources, and propose reengineering solutions,
- redesign: reengineer the BPs and activities,
- implement: pilot the BPs, train employees, implement full-scale, etc,
- evaluate: monitor the results, analyse searched goals, etc.

The well structured BPR methodology presented by Castano and al. [2] should also be mentioned in this section. In fact the general architecture of this methodology is based upon the ARTEMIS tool environment and is composed of four reengineering phases which are conceived to take into account requirements posed by distribution and heterogeneity aspects of BPs in complex companies [2]

In this paper we adopt the BPR consolidated methodology proposed by Muthu and al. [4] to model the process supporting it with UML diagrams because we argue that this modelling is important for tow reasons: (i) the comprehension of the BPR process aspects, through its modelling, will increase the success of the BPR operation, (ii) in the case of a failed BPR project, the process supporting this project can be itself subject of a BPR operation. In fact, the retained BPR methodology is composed of five main steps:

- ***step #1****: prepare for reengineering*, e.g. justifications of the need for the firm processes to be reengineered [11].
- ***step #2****: map and analyse as-is process*, e.g. understand and map the existing processes. The main objective of this phase is to identify disconnects (events that block the BPs from achieving desired results and in particular information transfer between organizations or people) and value adding processes. This is initiated first by the creation and the documentation of BPs models. Then, an estimation of the amount of time and cost (in terms of resources), that each activity requires, is calculated through simulation and ABC (activity based costing) [11]. This step is an iterative process.
- ***step #3****:design to-be processes*: e.g. produce one or more alternatives to the current situation which satisfy the strategic goals of the enterprise. This phase begins with benchmarking the best and innovative practices conducted in peer organizations in order to obtain ideas for potential improvements. Then to-be models are developed for reengineered BPs and new ones proposed. Finally simulation and ABC are performed to analyse the time and cost involved due to new changes.
- ***step #4****: implement reengineered processes*, e.g. develop a transition plan from the as-is processes to the redesigned BPs. The aim of this plan is to align the organizational structure, information systems and the business policies and procedures with the redesigned BPs. The transition plan is then validated using the simulation techniques and training programs for the enterprise employees are initiated. Finally the plan is executed in full scale.
- ***step #5****: improve process continuously*, e.g. monitoring the progress of action and the obtained results. The progress of action is measured by seeing how well the change teams are accepted in the broader perspective of the organization. Monitoring the results include such measures as employee attitudes, customer perceptions, supplier responsiveness [11]. In so doing, continuous improvement of performance is ensured through a performance tracking system and an application of problem solving skills such as adopting TQM to the newly designed BPs.

Taking into consideration the complexity of the BPR operation in one hand and the heterogeneity of the BPs to reengineer in a second hand, modelling the process supporting such projects will be very helpful. In fact we present in this paper, how to use the UML diagrams to fulfil this purpose, with a particular focus on the BPs modelling step in the retained BPR methodology.

4. Background Information on UML Modelling Constructs

The UML was developed in 1995 by Grady Booch, Ivar Jacobson, and Jim Rumbaugh at Rational Corporation, with contributions from other leading methodologists, software vendors, and users. Rational Corporation chooses to develop UML as a standard through the Object Management Group (OMG). The resulting co-

operative effort with numerous companies led to a specification adopted by OMG in 1997.

UML is a widely used standard object-oriented visual modelling language [17]. UML is also being used as a base object description language for the emerging UEML (Unified Enterprise Modelling Language) proposed by IFAC/IFIP [18]. In UML, there are three main modelling viewpoints, namely: use case models, static models and dynamic models. Use case models describe system requirements from user viewpoints. Static models are essentially class diagrams that describe system elements and their relationships (including generalisation, aggregation and association relationships). Dynamic models describe system behaviour over time [19].

4.1. UML Use Cases and Use Case Diagrams

Use cases define generic processes that the system must be able to handle. The building blocks of use case models are use cases, actors and the modelled system. The use case specifies the functionality provided by the system or a task that has to be done with support from the system. An actor is not a part of the system, but an external entity that must interact with the system. An actor is a type (a class), not an instance. Each type of actor represents a role, not an individual user of the system. Actors communicate with the system by sending and receiving messages. When an actor sends a message to a system this will initiate a use case[19] .

4.2. UML Class Diagrams

A class is a descriptor used to refer a set of objects with similar data structure, behaviour and relationships. A class diagram can be used to provide a static view of a system in terms of its object classes and the relationships among those classes. However, class diagrams do not encode temporal information. Four kinds of UML modelling construct (association, composition, generalisation and dependency) are used to describe static relationships within class diagrams.

4.3. Dynamic Modelling

In UML notation object interactions are described using a dynamic model. Modified system behaviour is normally initiated following the transmission of a message from one object to another. In UML, modelling constructs are provided to describe the four types of message namely synchronous, synchronous with immediate return, asynchronous and simple. UML also provides modelling constructs to construct four types of dynamic diagram, namely state diagrams, sequence diagrams, collaboration diagrams and activity diagrams. State diagrams and activity diagrams can be used to encode structural descriptions, whereas sequence diagrams and collaboration diagrams are designed to describe how behavioural descriptions are executed.

5. UML-Based Modelling Approach for the BPR Process

In this section, we show how the process supporting the retained BPR methodology [4] can be modelled using UML diagrams constructs, in order to suggest a systematic approach for the reengineering process in BPR projects.

In fact, we discuss the potentialities and the limitations of the UML for modelling the reengineering process and we introduce a systematic approach for the second and the third steps of this process.

5.1. UML Use Case Diagram of the Reengineering Process

The reengineering process mentioned above is composed of five steps. So, each step can be represented by a use case as follows [16]:

- use case prepare for reengineering : the CEO (Chief Executive Officer) articulates the strategic context and the justifications for the reengineering. In fact, the CEO defines the objectives, the scope and the approaches in initiating the BPR operation.
- use case map and analyse as-is processes : the PETs (Process Evaluation teams), before redesigning the BPs, should understand and create models of existing BPs in the company. Then in order to identify disconnects and value adding BPs, the PETs evaluate the amount of time and resources consumed by each BP through simulation and ABC.
- use case design to-be processes : the PETs benchmark the practices and performance of comparable organizations. Then, having identified the potential improvements to existing BPs, they develop to-be models bearing in mind the principles of process design. After several simulation and evaluation operations, the final, arrived at and validated, to-be models are selected for implementation.
- use case implement reengineered BPs : the LM (Line Management) is responsible of the development of a transition plan from the as-is to the redesigned BPs. So, the new information system supporting the new BPs is implemented, the new work procedures are applied and the workers are trained. The PETs are also involved in this step by communicating recommendations and building support and buy -in from the LM.
- use case improve processes continuously : this use case is initiated by the PETs because they regularly monitor BPs performance, benchmark emerging best practices and make continuing recommendations for BPs improvement.

Fig.1. illustrates our understanding of the reengineering process which is represented by the UML use case diagram as following.

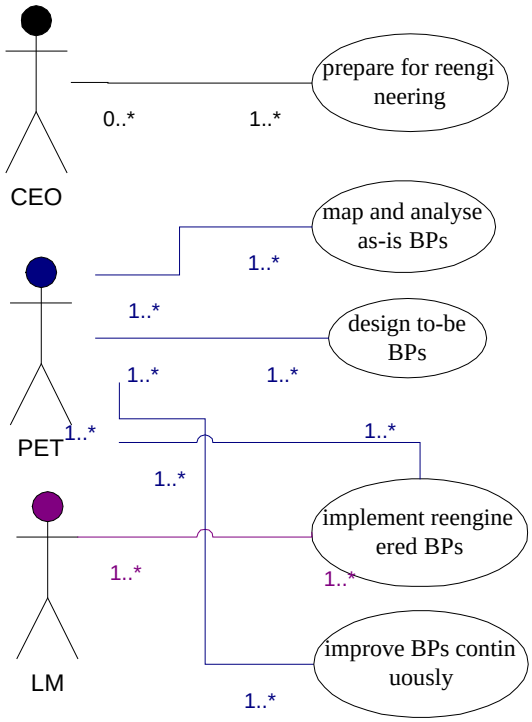


Fig. 1. UML use case diagram of the BPR process

5.2. UML Modelling-Based Approach for the Second and the Third Steps of the BPR Process

As mentioned above, we develop a systematic method for supporting the second and the third steps of the reengineering process and particularly the modelling steps. In fact, we are interested in these two activities because in [20] we are actually seeking for a generic approach for modelling BPs in BPR projects.

The UML-based approach consists in:

- for the *step #2*: two main sub steps are necessary :
  - *step #2.1- modelling the actual state of the existing BPs in the firm (as-is models)*-. This step can be realized using a top-down approach. Initially, a top level use case diagram describing the entire functionality of the organization is created. Then each use case, representing BPs, is refined in further use case diagrams and so on, as it is necessary. Each use case (i.e. BP or BP s activity) is described including the description of its workflow, its actors, its consumed resources, etc. So, implementation diagrams and class diagrams are produced for each use case. Moreover to complete the modelling of the actual state of the existing BPs, general BPs concepts should be identified and illustrated in further

UML diagrams (such as deployment diagrams, transition diagrams, etc.). In Fig.2 we represent the top-down UML approach for the fulfil of this step by using an UML activity diagram.

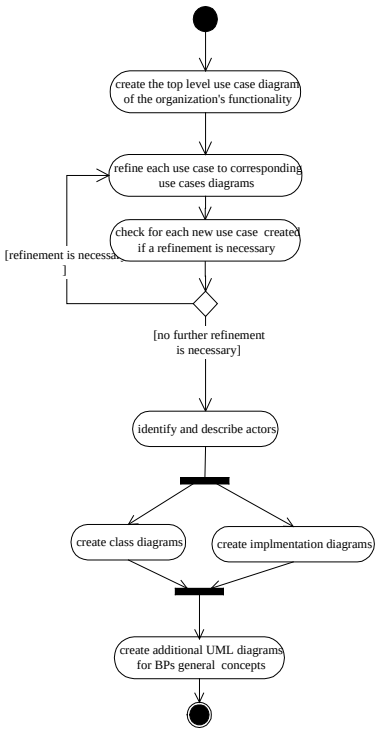


Fig.2. Step #2.1- modelling the actual state of the existing BPs (as-is models)

- *step #2.2- systematic identification of weak points, disconnects and value adding BPs-*. This step starts from the as-is BPs models results of the step #2.1, operates for simulation and ABC operations upon theses models in order to find out different aspects of the existing BPs (activity execution and coordination, exchanged information, employed organizational resources, etc.). Afterward, weak points are identified on the basis of the business objectives (service quality, productivity and efficiency, customer satisfaction, etc.) in order to select value adding BPs candidate for improvements. All the steps for this activity are represented with an UML activity diagram as it is illustrated in Fig3.

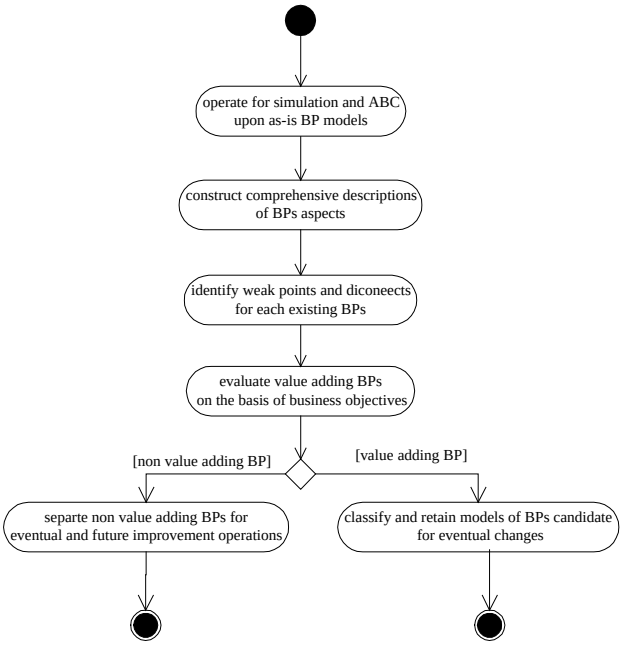


Fig.3. Step #2.2 - systematic identification of weak points, disconnects and value adding BPs.

- for the *step #3*: phases of this step are:
  - *step #3.1*: benchmarking the best practices in peer organizations,
  - *step #3.2*: designing to-be BPs and developing corresponding models,
  - *step #3.3*: simulating and evaluating performance of to-be BPs models in order to validate the retained ones for implementing improvements. If BPR objectives are not reached then a review of proposed changes for existing BPs is needed or a review of the proposed to-be models is also required.

In Fig.4 we give a representation of a top down approach for the fulfilment of the step #3 with an UML activity diagram.

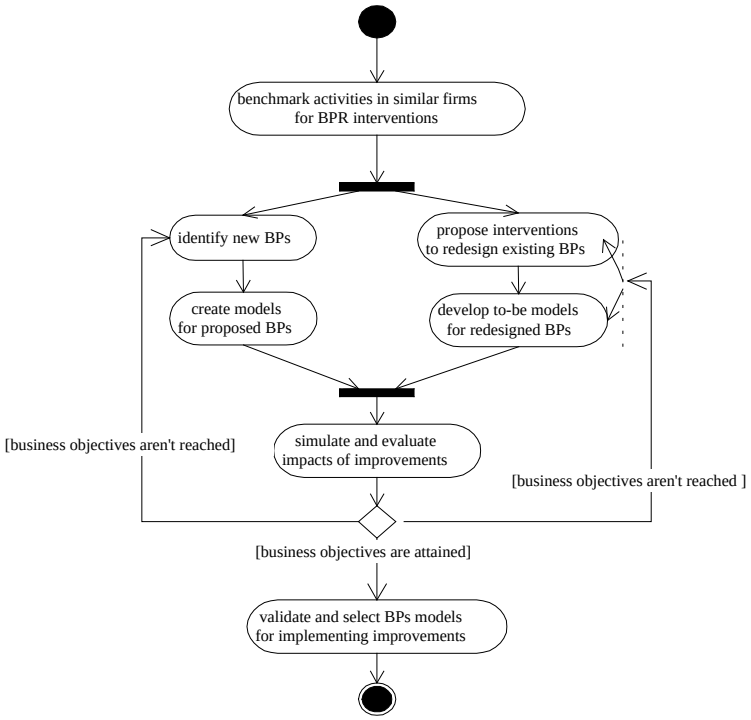


Fig. 4. Step #3- design to-be processes

5.3. Potentialities and Limitations of the Proposed Solution

UML is not the first modelling language used in the business community. In fact one of the primary notation to gain acceptance for business process modelling was the Integrated DEFinition (IDEF) [21]. The IDEF notation has enjoyed some succes in the business modelling community. However because many organizations adopted ad hoc/in-house notations, IDEF became too complex. Many other techniques are also adopted to model software processes and business processes, such as: ASME (American Society for Mechanical Engineerrs) [22], EPC (Event Process Chain) [23], EEPC (Extended Event Process Chain) [23], QPL (Quality Process Language) [24], I* Model (Distributed Intentionality Model) [25], ARMA(Agent Relationship Morphism Analysis) [26], etc.

The UML diagrams are also adopted for modelling several kinds of processes. Particulary it was used in modelling software processes, such as legacy systems, software-reengineering processes [27], etc. According to the case study presented in this paper, we argue that UML is adequate to model a broad range of systems (software processes, hardware systems, real-world organizations, business processes, reengineering processes, etc.). In fact, the different diagrams adopted to model the BPR process facilitate the comprhension, the application and the implemmentation of

this process. Once well defined and assimilated, the interaction between the actors and teams implied in the BPR project is being more easy and fluid. So, these later spend more time identifying the areas/activities of value creation ad those of value destruction. Once defined these areas can be addressed with specific strategies in order to increase the customer lifetime value.

Hence, the UML based approach for modelling the BPR process can not guarantee the success of the BPR project, but helps to increase its success rate. In fact, the proposed framework has been constructed with abstractions to many details and heterogeneous aspects of the BPR process: actors implied into, tools adopted for, strategic decisions adopted, etc. Neverthenless, this solution can be considered as a tool-support that provides framework in which practioners may focus more attention and researches.

6. Conclusion and Discussion

Business process reengineering (BPR) has been considered as an important way to reshape business organizations for achieving breakthrough improvements in performance. Several BPR methodologies are proposed in the literature but all with a primary focus on a need of a deep understanding of organization business processes (BPs) in order to obtain a comprehensive vision of the system and the evaluate possible restructuring interventions. So, creating a possibly representation of the process of reengineering itself will be very helpful for the success of the BPR project. In this paper, we propose an UML based approach for modelling the different steps of the BPR process. In fact based upon a BPR methodology given in [4], we adopt UML use case diagrams and UML activity diagrams to model the process of reengineering. Moreover more attention is given to the second and third steps and particularly the phases of modelling existing BPs and reengineered ones.

Adopting the different UML diagrams constructs in this case study; modelling the reengineering process, shows that UML presents several advantages allowing its use for broad range systems modelling (software systems, databases, real-time systems and real-world organizations).

Yet UML can not guarantee success in the BPR field, it establishes a consistent, standardised and tool-supported modelling language for a systematic modelling approach of the reengineering process in BPR projects, which can be itself a subject of a reengineering operation.

References

1. M. Hammer and J. Champy: Reengineering in the Corporation: A Manifesto for Business Revolution -Harper Business, 1993.
2. S.Castano, V.Antonellis and M.Melchiori :A Methodology and Tool Environment for Process Analysis and Reengineering . Data and Knowledge Engineering, 31, p: 253-278, 1999
3. D.Karagiannis, (Ed), Special issue on Business Process Reengineering, SIGOIS Bulletin, 16 (1), Août 1995.

4. S. Muthu, L. Whitman and S.H Cheraghi : Business Process Reengineering: A Consolidated Methodology . The 4th Annual International Conference on Industrial Theory, Applications and Practice. November 17-20, 1999. San Antonio, Texas, USA.
5. Y.C. Chen: Empirical Modelling for Participative Business Process Reengineering Thesis. University of Warwick. Conventry, United Kingdom.December 2001.
6. A. Martyn. Ould: Business Processes : Modelling and Analysis for Reengineering , Wiley,1995.
7. I. Jacobson: The Object Advantage . Addison-Wesley, 1995.
8. I. Bider : Business Process Modelling- concepts , Proceedings of the Practical Business Process Modelling: PBPM 00, 2000.
9. I.L. Wu : A Model for Implementing BPR based on Strategic Perspectives: an Empirical Study . Information and Management, 39,p: 313-324, 2002.
10. D.G. Wastell, P. White and P. Kawalek: A Methodology for Business Process Redesign- Experiences and Issues . Technical Report, Information Process Group, Department of Computer Science, University of Manchester, UK,1996.
11. T.H. Davenport and J.E. Short : The New Industrial Engineering: Information Technology and Business Process Redesign . Sloan Management Review 31 (4).p: 11-27. 1990.
12. T.R.Furey: Case study: Precision Materials ,Inc- A six step Guide to Process Reengineering . Planning Review, 21. p:20-23. 1993.
13. R.J. Mayer, P.A. Dewitte: Delivering Results: Evolving BPR from art to Engineering , 1998.
14. B.D. Harrison, M.D.Pratt: A Methodology for Reengineering Business », Planning Review, 21 (2), p :6-11. 1993.
15. R.L.Manganelli, M.M.Klein: The Reengineering Handbook: a Step by step Guide to Business Transformation . American Management Association, New York. 1994.
16. W.J.Kettinger , J.T.C. Teng, S. Guha: Business Process Change: A Study of Methodologies, Techniques and Tools . MIS Quarterly, 21.p:55-80.,1997.
17. R. Pooley, P. Stevens : Using UML Software Engineering with Objects and Components . Addison-Wesley, Reading, MA, 1999.
18. F.B. Vernadat, Avision for future work of the task force (IFACIFIP), http://www.cit.gu.edu.au/_bernus/taskforce/archive/
19. C.H. Kim, R.H. Weston, A. Hodgson, K. H. Lee: The Complementary Use of IDEF and UML Modelling Approaches . Computers in Industry, 50.p: 35-56, 2003.
20. L.Jamel.Menzli, S.Ayachi.Ghannouchi and H.Hadjami.Ben Ghézala: A Generic Approach for Modeling Business Processes in BPR (Business Process Reengineering) Projects IRMA2004-- the 15th Annual International Conference Information Resources Management Association- Innovations Through Information Technology- 23-26 Mai 2004- New Orleans Marriott- New Orleans, Louisiana, USA.
21. S.Kappes: Putting your IDEF0 Model to Work , Business Process Management Journal, vol3, no.2, p.p: 151-161. 1997.
22. J. Peppar and P. Rowland: The Essence of Business Process Reengineering , Prentice Hall, USA, p.p: 169-175. 1995.
23. H.kim and Y.Kim : Dynamic Process Modelling for BPR: A computerised Simulation Appraoch . Information and Management, vol 32, no.1, p.p: 1-13. 1997.
24. G.Born : Process Management to Quality Improvement , Wiley, p.p: 12-30. 1994.
25. E.S.K. Yu and J. Mylopoulous : Modelling the Organization: New concepts and Toold for Reengineering . www.cs.utoronto.ca/~eric/ieeeexp2.html
26. G.Valiris and M.Glykas: A case study on reengineering manufacturing process and structures , Knowledge and Process Management, vol.7, no.1, p.p: 20-28. 1996.
27. S.Garde, P.Knaup and R.Herold: Qumquad- a UML Based approach for remodelling of legacy systems in health care . International Journal of Medical Informatics-2003, volume: 70, p: 183-194.

Workshop 2.1:
Workshop on Evolution of Software Systems in a Business Context

Session:
Software Engineering Techniques and Evolution

Helping Conceptual Modelers Cope with Variability: An Illustration of Current Transformation Support

Jan Verelst
Dept. of Management Information Systems
University Of Antwerp
jan.verelst@ua.ac.be

Bart Du Bois and Serge Demeyer
Lab On ReEngineering
University Of Antwerp
{bart.dubois,serge.demeyer}@ua.ac.be

Abstract

Variability refers to the possibility of creating alternative correct models for a given set of requirements. Model variants have been shown to differ w.r.t. evolvability, thereby demonstrating the impact of making the right choice between variants. This paper suggests the application of current transformation techniques to support variability in conceptual models. When applied at conceptual models, such transformation techniques can help modelers cope with variability in two ways: (a) modeler's decisions can be more easily reversed; and (b) the equivalence of models can be validated by asserting the existence of a semantics preserving transformation. The former is helpful when the likelihood of evolution paths changes, and the latter is helpful in comparing and evaluating multiple models.

1 Introduction

Conceptual modeling is considered a so-called ill-structured design problem. This implies that different, correct conceptual models can be built for a given set of requirements in a certain modeling language. The possibility of creating alternative models is referred to as *variability*. We call different, correct models *variants* of each other.

Variability is problematic both for the conceptual modeler and for the end users in the business departments. First, because the modeler has to choose for one specific variant, thereby sacrificing the advantages of the other variants. There can be differences in, for example, evolvability of the model as illustrated in [Verelst, 2004a]. In other words, the conceptual modeler and the business user together decide which evolution path the model should favor, or still, which changes to business processes can be made quite easily. This demonstrates the impact of making the right choice between between variants. This choice is guided by the current prospection of likely evolution paths, which are bound to change over time. Second, variability is problematic if the end user is confronted with several models of his business requirements, because it is difficult to determine the differences and similarities in these models.

Techniques facilitating the transformation between model variants can help modelers cope with variability in two ways: (a) modeler's decisions can be more easily reversed ; and (b) the equivalence of models can be validated by asserting the existence of a semantics preserving transformation. The former is helpful when the likelihood of evolution paths changes, and the latter is helpful in comparing and evaluating multiple models.

This paper illustrates the current use of behavior preserving transformations to exploit variability. These transformations can be applied on any model with a clear semantical description, thus also on conceptual models. We discuss the current support, and lack thereof, for three types of variability identified in earlier work.

2 Existence of Variability

In an industrial setting, there is a high probability that different designers will build different conceptual models for a system. Possible explanations include differences in requirement interpretation, usage of synonyms for concepts or the addition of details. Moreover, variability also exists in a more controlled, restrictive setting, where different interpretations and other factors such as model layout are ruled out. Our conclusion so far is that variability is inherent to conceptual modeling and can occur in almost every modeling effort [Verelst, 2004b].

Recently, a non-exhaustive framework of three types of variability was proposed, based on a literature survey and empirical evidence [Verelst, 2004b]. The framework is aimed at variability in the initial identification of objects or entities in the requirements, and is applicable to object oriented and entity relationship modeling. The three types of variability are:

- **Construct variability**: Concepts are modeled using *different constructs of the modeling notation*. The semantics of the concepts are identical.
- **Horizontal abstraction variability**: Concepts are modeled using *different properties*.
- **Vertical abstraction variability**: Concepts are modeled *in a more abstract/generic or concrete way*.

Horizontal and vertical abstraction variability are quite fundamental, as they are linked with the translation of concepts in the real-world into classes or entities. This translation is a core activity of conceptual modeling. The framework therefore suggests that it is possible in virtually any modeling effort to create a variant.

3 Supporting Variability

In this section, we illustrate the application of existing transformations between variants of the three types of variability introduced earlier.

Refactoring – the application of structural modifications while preserving external behavior – is a transformation technique which is currently focused on the implementation level. A catalog of commonly applied refactorings has been composed, focusing on the redistribution of classes, variables and methods in the class hierarchy [Fowler, 1999].

In the research and practice of refactoring, the artifact being structurally modified is modeled as a graph [Mens et al., 2002b]. As refactoring is currently mostly applied at the implementation level, this graph is mostly a more convenient representation of the abstract syntax tree. Conceptual models can also be represented as graphs, and therefore, conceptual models are a potential target for application of refactorings. This application has already been demonstrated on UML class and state diagrams [Sunyé et al., 2001], and on database schema's [Delgado et al., 2003].

3.1 Construct variability

Construct variability refers to the possibility of modeling concepts in the UoD using different constructs in the modeling language. Some concepts in a UoD can be represented by a class or entity, a relationship, an attribute or even an instance.

We will illustrate the use of refactorings in two examples of construct variability.

- Transforming between an attribute and a class construct is supported by the Extract Class refactoring, which encapsulates a set of attributes and possibly also methods in a new class [Fowler, 1999].
- Transforming between a derived attribute and a method construct is supported by the Self Encapsulate Field refactoring [Fowler, 1999]. This refactoring replaces all accesses and updates to the attribute with calls to a newly introduced accessor and modifier method (respectively a getter and setter method).

These refactorings are commonly supported in todays Integrated Development Environments (IDE's).

3.2 Vertical abstraction variability

Vertical abstraction variability refers to the possibility of modeling concepts in the UoD in a more or less generic/abstract way. Parnas [Parnas, 1979] defines genericity as "software can be considered 'general' if it can be used, without change, in a variety of situations". Abstraction is defined as "a view of an object that focuses on the information relevant to a particular focus and ignores the remainder of the information" [IEEE, 1994].

Entire research directions, such as software patterns, software architectures and application framework are, at the analysis level, all forms of abstract models. Therefore, the transformation between a custom and a software design pattern implementation can illustrate the exploitation of vertical abstraction variability. Recently, it has been demonstrated how refactoring can support the introduction of design patterns [Ó Cinnéide, 2001, Kerievesky, 2004]. Examples of these refactorings are the transformation of an implicit tree structure to the use of the Composite pattern, or the replacement of conditional calculations with the Strategy pattern.

3.3 Horizontal abstraction variability

Horizontal abstraction variability refers to the possibility of modeling concepts in the UoD based on different properties. Like in vertical abstraction variability, the concepts in the UoD are modeled using different semantic definitions. In this case, the properties which are visible and localized can vary. These are called properties of the primary dimension. Properties of the secondary dimension, on the other hand, are delocalized and not visible.

As an example, consider the requirements for a school administration. It is possible to model concepts such that the properties of persons and their tasks are localized and visible. Alternatively, an equivalent model could model concepts such that properties of class rooms and courses are localized and visible.

Despite our active contributions to and knowledge on refactoring literature [Mens et al., 2002a], we are unaware of examples or applications in current refactoring literature exploiting this type of variability.

4 Conclusion

Variability occurs in almost every modeling effort, and its exploitation can help modelers in two ways: (a) modeler's decisions can be more easily reversed; and (b) the equivalence of models can be validated.

This paper suggests a research domain where proven transformations on code and design level are applied on the level of conceptual models. Currently, we are in the process of determining which changes have to be made to known refactorings in order to apply them in the most efficient way to conceptual models.

5 Acknowledgments

This work has been sponsored by the Belgian National Fund for Scientific Research (FWO) under grants 'Foundations of Software Evolution' and 'A Formal Foundation for Software Refactoring'. Other sponsoring was provided by the European Science Foundation by means of the project 'Research Links to Explore and Advances Software Evolution (RELEASE)'.

References

[Delgado et al., 2003] Delgado, C., Samos, J., and Torres, M. (2003). Primitive operations for schema evolution in ODMG databases. In *OOIS*, pages 226–237.

[Fowler, 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[IEEE, 1994] IEEE (1994). IEEE std 610.12-1990: IEEE standard glossary of software engineering terminology. In *IEEE standards collection: software engineering*. IEEE.

[Kerievesky, 2004] Kerievesky, J. (2004). *Refactoring To Patterns*. Addison-Wesley.

[Mens et al., 2002a] Mens, T., Demeyer, S., Du Bois, B., Stenten, H., and Van Gorp, P. (2002a). Refactoring: Current research and future trends. In *Language Descriptions, Tools and Applications (LDTA)*.

[Mens et al., 2002b] Mens, T., Demeyer, S., and Janssens, D. (2002b). Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag. Proceedings First International Conference ICGT 2002, Barcelona, Spain.

[Ó Cinnéide, 2001] Ó Cinnéide, M. (2001). *Automated Application of Design Patterns: a Refactoring Approach*. PhD thesis, University of Dublin, Trinity College.

[Parnas, 1979] Parnas, D. (1979). Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138.

[Sunyé et al., 2001] Sunyé, G., Pollet, D., Traon, Y. L., and Jézéquel, J.-M. (2001). Refactoring uml models. In *UML '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148, London, UK. Springer-Verlag.

[Verelst, 2004a] Verelst, J. (2004a). The influence of the level of abstraction on the evolvability of conceptual models of information systems. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 17–26, Los Angeles. IEEE CS Press.

[Verelst, 2004b] Verelst, J. (2004b). Variability in conceptual modeling. Technical Report Technical Report RPS-2004-019, University of Antwerp.

Knowledge Based Risk Management in Software Processes

Nicola Boffoli*, Marta Cimitile*, Aldo Persico[+], Aldo Tammaro[+]
**Dipartimento di Informatica – Università di Bari - Via Orabona, 4, 70126 Bari – Italy*
email: boffoli@di.uniba.it, cimitile@di.uniba.it
[+]EDS Italia Software S.p.A.-Via Antiniana,2/a, 80078 Pozzuoli (Naples) – Italy
email: aldo.persico@eds.com, aldo.tammaro@eds.com

1. Introduction

Unexpected events frequently cause major problems to projects. Furthermore, due to software project uniqueness, uncertainty about the end results will always accompany software development. While risks cannot be removed from software development, software engineers should learn to manage them better. Risk Planning requires organization experience, as it is strongly centred on the experience and knowledge acquired in former projects. The larger the experience of the project manager, the better his ability in identifying risks, estimating their occurrence likelihood and impact, and defining an appropriate response plan. However, project manager risk knowledge cannot remain in an individual dimension; rather it must be made available to the entire organization. Risk planning can be enriched by using knowledge and experience acquired by the various managers while working on the various organization projects. In order to do so, it is necessary for risk knowledge to be packaged and stored throughout project development, in order to use it in future. This is a challenging activity. The aim of this paper is to propose a knowledge based approach for project risk management. It is made of: a conceptual architecture for knowledge storing and reuse; a knowledge package structure for collecting risk knowledge.

2. Risk Management Process Pills

The aims of risk management process is to characterize the project by analysing the different project management functional areas (Communications, Procurement, Cost, Quality, Resource, Schedule, Scope) in order asses, point out and further manage the risks affecting a project.
The assessment phase is usually done by using focused questionnaires that support the project manager during risk discovering activity.
After having pointed out the candidate risk list, the project manager defines an appropriate response plan by using his/her own knowledge and experience [1, 2], in order to prevent, mitigate and face each risk.
At a glance, Risk Management can be seen as divided into two sub-processes [3, 4]: *Risk Evaluation* and *Risk Control*. *Risk Evaluation* is further divided into the activities *Risk Identification*, *Risk Analysis* and *Risk Prioritisation*. *Risk Control* is subdivided into the activities *Risk Planning*, *Risk Plan Integration* and *Risk Monitoring*. Each of the activities is then divided into sub-activities, where the use of organizational risk knowledge is a key factor. It is well known that reuse of previous experience for risk prevention is a key factor. Nevertheless, due to deadlines, high expectations for quality and productivity, and challenging technical issues, most software projects cannot dedicate the necessary resources for making their risk experience available for reuse.

Risk Evaluation	Risk Identification
	Risk Analysis
	Risk Prioritization
Risk Control	Risk Planning
	Risk Plan Integration
	Risk Monitoring

Figure1: Risk Management

A the same time people involved in the project execution need to reuse tailored risk knowledge and experience in order to address project critical success factors.

3. Proposed Approach

The proposed approach is made of two distinct components: a conceptual architecture and a proposal for a knowledge package structure for collecting risk knowledge.

3.1 Conceptual Architecture

In order to satisfy needs such as the ones previously mentioned, in the past an Experience Factory Organization (EFO) has been proposed [5, 6, 7, 8]. EFO distinguishes project responsibilities from those related to collection, analysis, packaging, and experience transfer activities. In doing so, it forecasts two different organizational units: Project Organization (PO) and Experience Factory (EF). The first uses experience packages for developing new software solutions and the second provides specific knowledge ready to be applied. As support to these two infrastructures and as completion of the EFO approach, a paradigm for continuous quality improvement, Quality Improvement Paradigm (QIP) is also used [5].
The authors of the present paper agree on the usefulness of this model that, although well known in the community, is not supported by sufficient empirical evidence concerning its use. In fact, it is a general model that requires to be specialized in operative terms, as it has

been done in [5]. Nevertheless, many experiences are reported in literature.
The phases that make up the presented model are briefly described in the following [5]:
Learn: requires classification and analysis of all the partial artefacts produced during project execution, and knowledge packaging. In this phase new knowledge packages are defined or existing ones are updated. Furthermore, in this phase the knowledge collection and fruition process is improved.
Plan: requires characterization, description of project goals and choice of the strategy to adopt. It consists in reusing experience collected and acquired in previous projects and creating knew knowledge. Therefore, this phase selects available knowledge packages that can most likely be reused in the project being carried out.

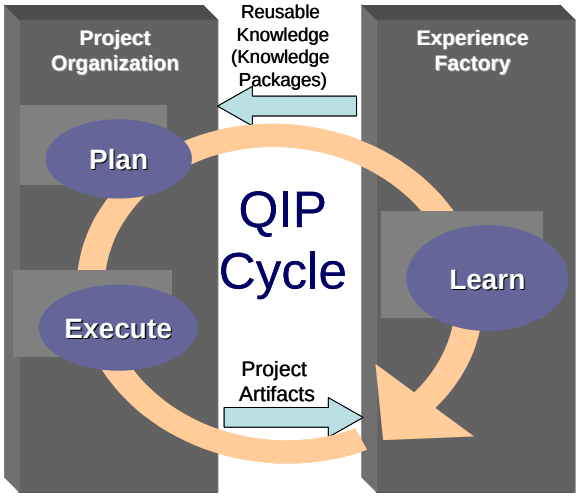


Figure 2: QIP Cycle

Execute: in this phase the selected packages are applied and all project data and produced artefacts are collected.

3.2 Risk knowledge package structure

The EFO approach is generally valid regardless of the way knowledge is represented. Nevertheless, its specialization in an operative context requires it to be tailored by using a specific formalization approach.
Knowledge faced varies from typical document templates, spreadsheets for data collection and analysis, project documents etc. Such a document base can be used each time one desires analyzing what has occurred during project execution in order to reuse part of it or make decisions according to what has occurred. In this work, an innovative approach for knowledge package definition has been added to what has previously been mentioned. The proposed approach is based on use of decision tables [9, 10, 11].
In particular, a set of decision tables have been used to formalize knowledge first and then provide it. Knowledge refers to: project characterization, explication of relations among the characteristics, risks encountered during project execution and the most appropriate response plan for mitigating and facing risk.

So, given these considerations, the authors provide some details on how to use decision tables. For space reasons we will not refer to the structure of other knowledge packages defined for document management.

A decision table is a tabular representation of a procedural decision situation, where the state of a number of conditions determines the execution of a set of actions [12]. In general, a decision table is a table divided by a double line, horizontal and vertical, into four quadrants Figure 3. The horizontal line divides the table in a conditional part (the top part) and an action part (the bottom part). Moreover, the vertical line divides the input values (left side) from the rules and combination of conditional states (right side). The table is defined so that each combination of conditions (conditional states) corresponds to a set of actions to carry out (rule). A tabular representation of a decision situation is characterized by a separation between conditions and actions on one end, and between rules and conditional expressions on the other. Each column of the table (decision column) identifies which actions should (or shouldn't) be carried out for a specific combination of conditional states. The conditional oriented approach of a decision table allows to express all the knowledge related to the problem being considered (in our case the identification of risk driver together with the response plan to use in order to face the risks.).

CONDITIONS	CONDITIONAL STATES
ACTIONS	RULES

Figure 3: Quadrants of a Decision Table

Following to the generic presentation of the decision tables, we now focus our attention on presenting, through an example, how they have been used for representing risk knowledge.

1. Project attribute 1	L									M									H									
2. Project attribute 2	L			M			H			L			M			H			L			M			H				
3. Project attribute 3	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H	L	M	H
1. Risk Driver x	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x	x	x	x	x	x			
2. Risk Driver Y	.	.	.	.	.	.	x	x	x	.	.	.	.	.	.	x	x	x	.	.	.	.	.	.	x	x	x			
3. Risk Driver z	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x			
4. Mitigation Action 1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	x	x	x	x	x	x	x	x	x			
5. Mitigation Action 2	.	.	.	.	.	.	x	x	x	.	.	.	.	.	.	x	x	x	.	.	.	.	.	.	x	x	x			
6. Mitigation Action 3	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x			
7. Mitigation Action 4	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x	.	.	x	x	x	x	x	x	x	x	x	x			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27			

Figure 4: Example of a Decision Table

In the CONDITION quadrant we have project characteristics (for example cost, time, available personnel etc.); in the CONDITIONAL STATES quadrant we have the possible values of project characteristic (typically a Low, Medium, High scale); in the ACTION QUADRANT we have the risk driver that must be faced (for example schedule, scarceness of personnel, etc.) together with the possible mitigation

action to carrying out (for example increase the number of human resources allocated on a project, define a new date for project conclusion, etc). Finally in the RULES quadrant we have the relationship between project characteristics, risk driver and correspondent mitigation action to carrying out.
It is clear that the previously described structure allows to verify the effectiveness of the executed actions and to consequently extend and update previously acquired knowledge. For example, if a mitigation action were to be ineffective it can be deleted from the knowledge package; project characterization can be enriched by adding new context parameters and so on.

Conclusions

In the present position paper we have proposed a new approach for risk management in software projects based on the collection, formalization and use of previously acquired knowledge. This approach consists of a conceptual architecture represented by the EFO model and of an approach for formalizing knowledge based on decision tables. At the moment we are experimenting the described approach in an industrial project carried out in collaboration with EDS Italia Software. In particular, we are packaging both EDS managers' experiences and that extracted from executed projects, through decision tables. In the next phase the approach will be experimented during real project execution.

Bibliography

[1] Markkula M., Knowledge Management in Software Engineering Project, Software Engineering and Knowledge Engineering, SEKE 99, Kaiserlautern, June, 1999.

[2] Startz J., Leverage your Lessons, IEEE Software, Volume 16 , Issue 2, March 1999.

[3] Boehm B., Tutorial: Software Risk Management, IEEE Computer Society, 1989. .

[4] Farias L., Travassos G. H., Rocha A. R., Managing Organizational Risk Knowledge, Journal of Universal Computer Science, July, 2003

[5] Kontio J., Basili V. R., Risk Knowledge Capture in the Riskit Method, Computer Science Technical Report, 1996.

[6] Basili V. R., Software Development: A Paradigm for the Future, Proceedings of the 13th Annual Computer Software and Applications Conference (COMPSAC), 1989.

[7] Basili V., Caldiera G., Rombach H., The Experience Factory, John Wiley & Sons, 1994.

[8] Basili V. R.,. Caldiera G, McGarry F., Pajerski R., Page G., Waligora S., The Software Engineering Laboratory - an Operational Software Experience Factory, Proceedings of the International Conference on Software Engineering, May 1992.

[9] HO Tu Bao, Introduction to Knowledge discovery and data mining, Institute of Information Technology National Center for Natural Science and Technology.

[10] Vanthienen J., Mues C., Wets G., Delaere K., A tool-supported approach to inter-tabular verification, Expert Systems with Applications, 1998.

[11] Maes R., Van Dijk J. E. M., On the Role of Ambiguity and Incompleteness in the Design of Decision Tables and Rule-Based Systems, The Computer Journal, 1988.

[12] Pooch U.W., Translation of Decision Tables, Computing Surveys, June 1974.

An approach to guidance in Extreme Programming project management

Houda Zouari Ounaies[1], Yassine jamoussi[2], Mohamed Ben Ahmed[3]
ENSI, National School of Computer Science, 2010, Manouba, Tunisia
[1] Houda.Zouari@isetr.rnu.tn [2] Yassine.jamoussi@ensi.rnu.tn
[3]Mohamed.BenAahmed@riadi.rnu.tn

Abstract

In recent years, a great interest has been advocated to agile methods, particularly, eXtreme Programming or XP. This method creates controversy critics: Some experts consider that XP include essential engineering practices for every software project. Others consider that these practices are impracticable and go against project development productivity.

We consider that XP is based on practices of good sense and can give many benefits to organization adopting it. However, we consider that the principle lack in XP is project management and especially the measurement process. This issue is non-sufficiently defined in XP. We propose in our approach to improve XP method taking into consideration Capability and Maturity Model Integrated (CMMI) objectives. We focus on measurement process and we provide guidance for its formalization. Improved XP process is critiqued from Capability and Maturity Model for software (SW-CMM) perspective. This model considers measure as a key concept. We think that proposed improvement allow organization adopting XP to reach SW-CMM level 3 and to address some of level 4 practices.
Key words: eXtreme Programming, software development, process improvement, project management, measures, CMM, CMMI, guiding, experience capitalization, feedback, Case-based reasoning.

1 Introduction

Extreme Programming (XP), an agile method officially burned in 1999, creates controversy critics in software engineering communities: Some experts consider that most of XP consists of good practices that should be thoughtfully considered for any environment [1]. Others experts consider Extreme Programming to be harmful for reliable software development [2].

We share the opinion of experts who think that XP can guaranty the success of small or medium development projects if certain issues of XP are improved. In this article, we propose XP improvement guidance. This improvement can lead organization adopting XP to reach high level of the capability Maturity Model (CMM) for Software.

2 Extreme Programming

XP method dimensions one development project by four variables: cost, time, quality and scope. One XP team control the three first variables by regulating its work rhythm basing on the scope variable and by adopting iterative cycle, basic practices and basic values.

2.1 XP practices

XP Basic practices are:

1. Programming practices: *Simple design, refactoring, automated test (acceptance and units*),

2. Collaboration practices: *Pair programming, Collective ownership of the* code, *Coding standards, Metaphor, Continuous integration (*many times a day),

3. Management project practices: *Small releases* on a very short (two week) cycle, *iterative planning, On-site customer, 40-hour week (*never work overtime two weeks in a row).

2.2 XP values

Four values are essential in an XP project:

1. Simplicity -- developers should adopt the simplest solution work,
2. Communication -- between team members,
3. Feedback -- essentially by automated tests,
4. Courage – to admit problem and to face them.

2.3 XP cycle

XP project cycle consists of two embedded cycles: releases and iterations (figure 1).

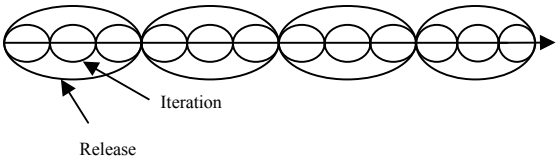


Figure 1. XP Project cycle

Release cycles concern functionalities visible by client. At the cycle beginning, releases are planned. Control is carried out by measures such us *failed acceptance tests rate and defects number. At the* cycle end, release is made even if not all scenarios planned have been developed. Implementation is done in a success of iterations. Iteration cycle concerns tasks realized by developers. At the start of iteration, these tasks are defined with client presence. Monitoring is carried out by indicators presented by measures such us failed units tests. At the end pf iteration, a table of principle indicators is displayed to all the development team.

3 XP process critics

XP present practices of good sense, which place developers and client in the centre of development process. Nevertheless, we have noticed that guidance in problem resolution during XP project is not explicit, even if XP practices aims essentially to reduce risk in software development project. XP does not emphasize process definition or measurement to the degree that models such as the CMM (capability Maturity Model) do [1]. Thus, XP's "activities" are not formally identified or described [3]. Relations between problems, reflected by measures, and solutions, concertedly implemented by XP practices are informal.

It is inherently difficult to manage what cannot be measured objectively [4]. Therefore, measurement is actually considered as a key practices to control, improve the development process and software quality. In this work we propose to improve this facet of XP project management process by formalizing XP measurement process. The result is an improved XP process that allow organizations adopting XP to control project efficiently and therefore to resolve the problems on time and even to anticipate them.

4 XP process improvement approach

The guidance that we propose to improve XP measurement process is based on ISO/IEC 15939 [5] Software Measurement Process, Practical Software and Systems Measurement and the SEI CMMI [6].

The CMMI (Capability Maturity Model Integrated), a Software Engineering Institute (SEI) evaluation and improvement model dedicates a key process area (KPA) for measurement: Measurement & Analysis. This KPA is based on ISO/IEC 15939 Software Measurement Process. The PSM (Practical Software and Systems Measurement) [7] is an implementation of this standard. PSM process model is presented in Figure 2.

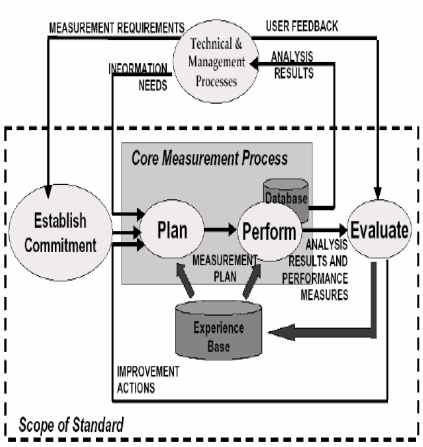


Figure 2. PSM process model

We propose to adopt PSM guidance in XP measurement process formalization. We summarize In table 1 the mapping between XP measurement process and PSM process model.

PSM process model phases	XP measurement process conformity
Commitment establishment	**Largely adopted***:* XP adopt measures.
Measurement planning	**Partially adopted:** In XP, it is not formally based in information needs of decisions makers and teams.
Measurement perform	**Partially adopted:** Project measures are not formally structures in a database
Measurement evaluation & improvement actions	**Partially adopted**: This activity is based on user feedback. In XP, feedback is possible thanks to practices such us frequent releases and automated acceptance and units tests. But this feedback is not formally based in a project measured experience (the database of the project) and organizational measured experiences (Experience Base)

Table 1. Mapping between XP measurement process and PSM process model

The mapping between XP measurement process and PSM process model permit us to consider two principles lacks in the actual XP method guidance:

1. XP measurement planning should be based on information needs of decisions makers and team

2. XP Measurement evaluation and Improvement actions should be based on measured feedback reached by getting XP project measured experience and XP organizational measured experience.

In this article, we deal with the second issue concerning XP measured feedback. We propose to use data mining approach in order to formalize the process of XP measurement evaluation and improvement actions. Various tools are associated to this approach, such as Neural Network, Decision Tree, Case-Based-Reasoning (CBR). The choice of one specific technique leads us to do a compromise between results prediction and results readability. Communication and interaction in the team are essential in XP and in agile environment generally. Thus, we propose to use CBR approach which is characterized by the extreme readability of their results.

4.1 CBR XP measurement process formalization

Capitalization of passed project experience is an important factor to guide current project. Indeed, feedback acquired by teams during development process is constantly improved and better interpreted. This will allow teams to improve their aptitude to evaluate the project situation and to take effective decisions. In XP project management context, we propose to use CBR systems. Problems are resolved by comparing *the situation* of the actual project to others cases stored in the “Case Base”. In the PSM model, “Case Base” is referenced as “Experience Base”. If one passed experience is sufficiently similar to the actual case, *solution* adopted in the passed experience can be applied to the current situation [8].

XP software project case is characterized by its situation and the experimented solution. We propose to structure them by hierarchic concepts. The XP project situation is described by its basic characteristics (type and size), its advancement and by different measures performed measures during the development process (figure 3).

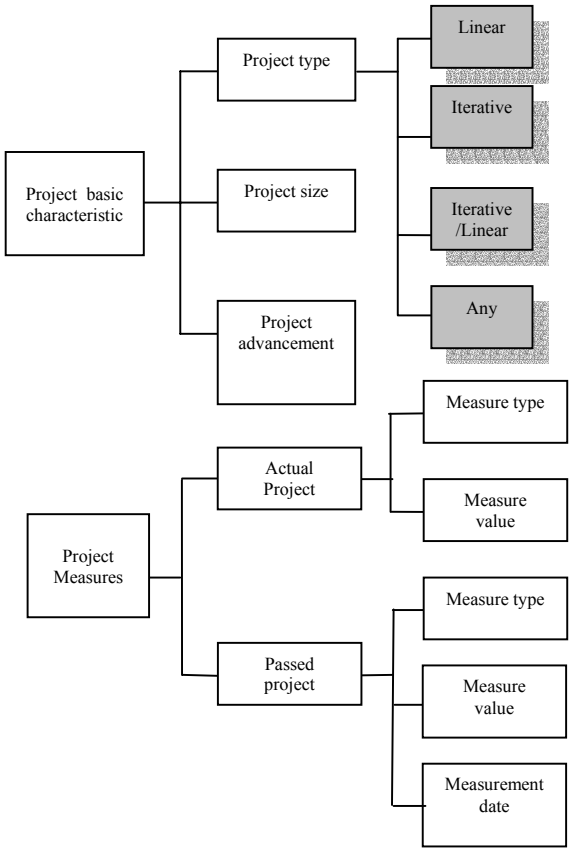


Figure 3. CBR XP structured situation

XP experimented solution stored in the "Case Base" is structured essentially by improvement actions used during XP process and by the evaluation of results obtained after using these actions (figure 4).

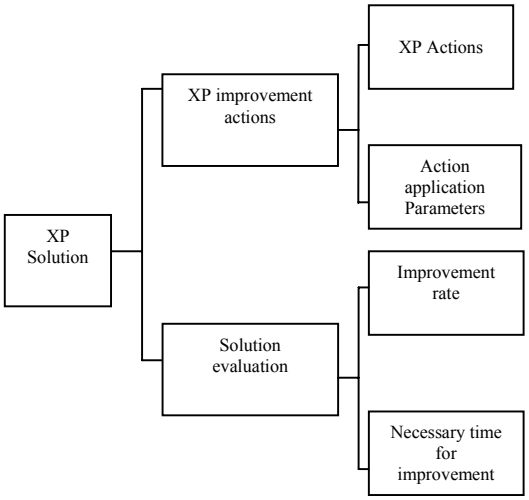


Figure 4. CBR XP structured solution

Nevertheless, we have met some problems during technique deployment. In deed, extracting knowledge from experiences of others projects necessitate an important Case Base which is not always possible. An other problem concerns results proposed by RBC systems. In fact, the latters favour results readability more than their predictability. Data-mining combination techniques can eventually bring to the capitalization process a better compromise between predictability and legibility in proposed solutions.

4.2 XP process modelling

We consider that modelization is a complementary solution in our XP process improvement approach. Modelling XP process allows its institutionalisation all over the organization. Moreover at CMM level 3, Organization Process Definition and Software Product Engineering key process areas essentially mean having a process model [9]. In this context, we propose to use the Software Process Engineering Meta-model (SPEM) [10], in order to model improved XP management process. SPEM is an OMG meta-model. We chose it since it represents different interactions between actors in the process.

The modelling result is represented by figure 5. XP measures considered in this model are:

1. Anomalies and incidents rate,
2. Regression rate,
3. Acceptance tests failure rate,
4. Technical delay,
5. Communication with client,
6. Team cooperation,
7. Etc.

Communications with client and team cooperation are qualitative measures, so we propose the assumption of quantitative values to each qualitative measure.

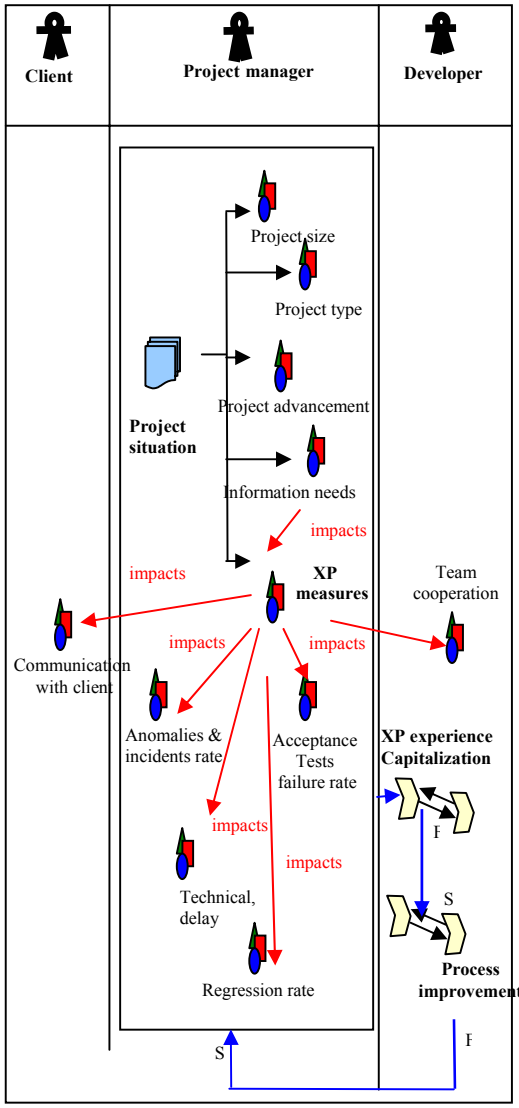


Figure 5. XP improved process model

4.3 Evaluation of XP improved process from CMM perspective

Formalizing XP measurement process according PSM model provides guidance for an effective XP management process. Our evaluation of XP improved process is based in a previous evaluation of XP from SW-CMM elaborated by Mark Paulk [1].

All CMM models and in our context SW-CMM give much importance to measurement in the process: Measurement concepts are spread all over SW-CMM Key process area. Therefore, we think that the formalization of XP measurement process from a CMMI perspective will improve results obtained in the earlier XP evaluation from SW-CMM perspective. We resume this evaluation in Table 2.

We think that adopting eXtreme Programming with a formalized XP measurement process allow an organization adopting this method to improve its development process reach the level three and to touch the level 4 of SW-CMM model.

SW-CMM maturity level	Actual XP process satisfaction	Improved XP process satisfaction	Improvement proposed solutions
Maturity level 2 : Repeatable			
Requirement management	++	++	--
Software project management	++	++	--
Software project tracking and oversdight	++	++	--
Software subcontract management	--	--	XP is dedicated to small and medium project
Software quality assurance	+	++	Guidance for a formalized measurement process conformed to ISO/IEC 15939 norm.
Software configuration management	+	++	Project experience capitalisation.
Maturity level 3: Defined			
Organization process focus	+	++	- Institutionalisation by XP improved process modelling, - Formalizing XP measurement process : the strenghs and weakness of the software process used are identified relative to a process standard.
Organization process definition	+	++	- Guidance in project management process, - Modelling improved XP process.
Training program	--	+	Organisational experience capitalisation
Integrated software management	--	--	--
Software product engineering	++	++	--
Intergroup coordination	++	++	--
Peer reviews	++	++	--
Maturity level 4: Managed			
Quantitative process management	--	+	Project management guidance by using measures
Software quality management	--	--	--
Maturity level 5: Optimized			
Defect prevention	+	+	--
Technology change management	--	--	--
Process change management	--	--	--

++ Large satisfaction + Medium satisfaction -- Low satisfaction

Table 2. Evaluation of XP improved process from SW-CMM perspective

5 CONCLUSION

Adopting XP, a controversy agile method, can bring much good practices of good sense. However, we consider that project management is non sufficiently defined in this method and particularly the measurement process.

Measures are critics to control, monitor and improve software development process. In this work we have proposed guidance to XP measurement process formalization. This formalization is in two steps. The first step consists to adopt PSM guidance. We noticed that formalizing XP measurement process leads to formalizing the measured experience, at project and organizational levels. We have proposed to use Case Based Reasoning (CBR) approach to do this. The second step of the formalization consists of modelling of XP measurement process improved. Finally we have evaluated the XP improved from a CMM perspective. We notice that such improvement would lead an organization to reach the level three of SW-CMM and to touch the level four.

Improving the maturity of the organization development process leads to control improvement of cost, time and quality in the software project. High maturity is reached only by some organizations in all the words. And we think that more researches are needed to assist organizations to improve their capacity to produce software.

References

1. Paulk, M. C., (2001). "Extreme Programming from a CMM Perspective." *IEEE Software*.November/December, pp. 19-26.

2. Extreme Programming Considered Harmful for Reliable Software Development, Gerold Keefer, AVOCA GmbH, 2002. On-line at:

   http://www.agilealliance.com/articles/articles/XPConsideredHarmful-GeraldKeefer.pdf

3. John Smith, "A Comparison of RUP and XP." *Rational Software White Paper*, May 2001.

4. Walker Royce, Software Project Management: A Unified Framework. Addison-Wesley, 1998.

5. ISO, ISO/IEC, "IS 15939: Software Engineering – Software Measurement Process Framework," International Organization for Standardization, Geneva, 2003.

6. CMMI Product Development Team, *CMMI[SM] for Systems Engineering/Software Engineering, Version 1.02 (CMMI-SE/SW, V1.02) Staged Representation*, Carnegie Mellon University, CMU/SEI-2000-TR-028, ESC-TR-2000-093, November 2000.

7. Dept. of Defense and US Army, "PSM: Practical Software and Systems Measurement – A Foundation for Objective Project Management," Version 4.0c, March 2003.

8. Kolodner, J. 1993. Case-Based Reasoning. San Mateo, CA: Morgan Kaufmann.

9. Suen Vivienne, Process modelling and automation, presented to OSQA/ASQ SWFG, Ottawa, 2004-03-16. On-line at: www.osqa.org/documents/suen.pdf.

10. OMG. Software Process Engineering metamodel Specification; adopted specification. Object Management Group, 2002.

Workshop 2.1:
Workshop on Evolution of Software Systems in a Business Context

Session:
Design Techniques Coping with Evolution

Improving Web applications Evolution by Separating Design Concerns

Gustavo Rossi[1,2], Silvia Gordillo[1,3] and Damiano Distante[4]

[1] LIFIA. Facultad de Informática. UNLP. La Plata, Argentina
{gustavo,gordillo}@lifia.info.unlp.edu.ar, http://www-lifia.info.unlp.edu.ar
[2]Also CONICET, [3]Also CICPBA
[4]RCOST - Research Centre on Software Technology. University of Sannio, Italy
distante@unisannio.it, http://rcost.unisannio.it

Abstract. In this paper we discuss how to improve the evolution of Web software by using different strategies for separation of design concerns. We first motivate our work by briefly presenting the different kinds of evolution problems that software developers face when dealing with business Web applications. Then we briefly describe our experience in the conception of a mature design method, the Object-Oriented Hypermedia Design Method (OOHDM), and in the development of a design model for designing business processes in Web applications (UWAT+). We show how we have dealt with design problems by using different abstraction and decoupling mechanisms and analyze the impact of our design choices in the evolution of target Web applications. We finally discuss some further work on this subject.

1 Introduction. The problems of Web Applications Evolution

Developing complex Web software is a challenge; the evolution of the WWW has proved to be quite fast and nowadays the Web is a platform for commerce, learning, government and other applications domains. Web applications must provide timely data and personalized behaviors, customizable interfaces and ubiquitous access; security and privacy are further issues to be considered.

As Web applications combine the power of unstructured and navigational management of multimedia data typical of hypertext systems, with other more conventional transactional behaviors, Web modeling and design approaches must cope with a myriad of problems. Some of them are typical of "conventional" software while others are idiosyncratic of the Web, as for example defining the units and paths of navigation, the relationships among web pages and underlying data, the interaction between navigation and other application behaviors, the design of adaptable (and adaptive) interfaces, etc. An additional challenge, addressed in this paper is how to deal with the evolution of these applications. Understanding the evolution patterns of Web applications is a key to find a good strategy to manage the process in a reasonable way. While development processes and associated tools are important for taming

evolution, what we need is to use good and adequate abstraction and modularization mechanisms to clearly separate design and development concerns that might evolve independently: we think that views, aspects and roles are some of the most important ones in the designer's armory.

Suppose for example the process of engineering a complex e-commerce application like www.amazon.com (which in fact has undergone a process of constant evolution), which different kinds of evolution must we face? We can, for example, add new stores to our site (Amazon began with 2 stores and now hosts 34), provide new functionality such as wedding lists, improve navigation by adding new links (types) between products, add new kind of offers, personalize the site, simplify the associated business processes (such as the check-out process), etc. It is easy to see that the main difference between all these examples is to which application concern they belong to.

In this paper we reflect on our experience both in the engineering of complex Web software and in supporting the evolution of these applications. We briefly show how we dealt with different types of evolution problems by emphasizing the impact of separation of concerns with respect to the ease of evolution.

Even though our presentation is based on design issues pertaining to the use of the Object-Oriented Hypermedia Design Method (OOHDM) [20, 21] and for the UWAT+ approach for modeling business processes, most of the ideas can be easily generalized to other similar methods, such as the Unified Web Engineering (UWE) [9, 10]. In Section 2 we describe the abstraction mechanisms used in OOHDM and in Section 3 we reflect on business process issues, using UWAT+ as an example.

2 Separation of Concerns in OOHDM: Views, Roles and Aspects to the rescue

2.1 Background

As other design methods for Web applications such as WebML [4], UWE [9, 10] and WSDM [5], OOHDM [20] partitions the design space in four activities: conceptual and application modeling, navigation design, abstract interface design and implementation. In this paper we emphasize those evolution patterns that impact on the conceptual and navigational model, by studying the relationships among the conceptual and navigation model and some design issues specific to the navigation model. The conceptual model aims to describe the domain semantics (data and behavior) in terms of classes, using the UML notation [6]; navigation issues are ignored in this model, though relationships will be later used to describe links

Meanwhile, in the navigation model we specify the nodes, links and other navigation structures (such as indexes and navigation contexts) that will be perceived by the user. Finally the interface model describes how navigation objects will be perceived by the use (according to his interface device and preferences). These concerns are decoupled (and connected) by the use of views as the main formalism. Nodes have usual hypertext semantics and are considered views on conceptual objects; links de-

fine the navigation topology and are considered views on conceptual relationships. Interface objects are meanwhile views on Nodes. The rationale for choosing this mechanism is that the relationships between nodes and objects (and similarly between links and relationships) follow closely the ones in the Observer pattern [7].

Nodes can "observe" more than one object; this fact allows us to build nodes opportunistically by "copying/pasting" different attributes of conceptual objects by using a simple query language that "navigates" conceptual relationships. In the popular Amazon site, the node standing for the CD "How to dismantle an atomic bomb" (an instance of the node class CD) contains information from the corresponding compact disk (CD) and information from other objects such as the name of the performing group (U2, an attribute of the object corresponding to Class Artist). Additionally, different Observers can be built for the same objects which allow describing different navigation structures according to the user role or profile. For example we will specify different CD Node classes for the customer or the manager user roles. Views are the best mechanism for coping with these problems.

Meanwhile, the use of roles as an alternative to classification, to express more dynamic assignment of behavior to objects has been early proposed by Pernici [13] and later discussed in [11] and formalized in [23].

Roles can be used in a straightforward and natural way to describe variability in nodes according to the incoming link used to reach the node. In fact, this application of the role concept clearly reflect the seminal ideas of Sowa [22], and also in [23] with respect to the role of roles to characterize how objects "look like" (e.g. behave) when involved in a relationship; and certainly hypermedia (the domain of navigation) is a paradigm in which relationships are first class citizens.

In [17] we have characterized the different possible uses of the role concept for expressing navigational variations. In [18] we used role models [16] to build full-fledged navigational (and conceptual) structures according to a theme of interest in the application. For example we can browse a virtual museum interested in its history, in the artworks, etc.

Web applications clearly involve different cross-cutting aspects, most of them archetypical, like persistence, security, logins, etc. Though we have not used aspects in OOHDM so far, it is not difficult to imagine how our method can benefit from several ideas coming from the Aspect fields. From the point of view of an OOHDM navigational model, an interesting use of aspects would be to capture the user navigation activity, for example to provide him information about the pages he visited. One way to provide this information (as in Amazon) is to add code such that each time we navigate to a page the page is included in the list. This code clearly cross-cut all classes and specifying it as an aspect would allow us to treat it separately. Aspects could be also used to describe extensions to nodes, even when they do not involve extending the conceptual model. Extending the functionality of nodes (See for example the "Better together" section in an Amazon buying offer) is usual and many times this kind of functionality is just temporary (it evolves to permanent when it is successful). Describing extensions with aspects helps to keep the base node classes more stable. Issues related with context-aware navigation (such as in Adaptive Hypermedia) can be also considered aspects as mentioned in [2].

2.2 Impact on Evolution

The use of objects to improve software evolution has been widely discussed in the literature, so we do not address it here. We mainly focus on these issues which pertain to specific aspects of Web applications. Therefore, we omit to discuss the evolution of the conceptual model and focus on navigational issues.

The most important decision of the OOHDM design approach is the separation of navigation from conceptual objects; this separation allows that the navigational object can evolve independently of the existing objects. Portals (e.g. home pages) can be built as aggregations of views of existing conceptual objects, i.e. we don't need to modify the conceptual model when wanting to extend a home page with new information (e.g. showing novelties, recommendations, etc).

For example, when new links must be described (e.g. when building richer navigational structures), we only need to specify the newly defined links, and eventually provide new methods (in existing application classes) to calculate end points when needed. There is no need to edit working code. Modification in the nodes' contents can be also made by just changing the definition of the corresponding Node class without interfering with existing code.

Roles allow enriching navigation seamlessly as they allow designing slight variations of the way nodes look like, according to the navigation path. Roles act as Decorators [5] and thus help to extend class functionality (in this case a Node's class functionality) dynamically without modifying the base class, and without even the need to define sub-classes.

Finally, though not completely explored in the Web engineering arena, aspects can be used to design volatile extensions, e.g. those new functionalities that are not completely consolidated (e.g. up to the moment they are tested with users). The use of aspects to design extensions is relatively new but is a key approach for coping with unexpected (and even transitory) evolution.

3 Separation of Concerns in UWAT+: Business Process Activities and Content Navigation

Aspects, roles and views help separating design concerns at design time and ease the evolution of Web applications during maintenance time through the evolution of the portion of the design of interest, which is likely to involve a more circumscribed part of the application code.

Though not explicitly stated, separation of design concerns is also adopted in UWAT+[27][28], an extended and revised version of the Ubiquitous Web Application (UWA) transaction model [25][26] specifically tailored for designing business processes in Web applications.

In the following of this section we'll give an overview of the UWAT+ model and design approach, recognizing separation of design concerns in its conception and how this may help in evolving the designed application.

3.1 Background

UWAT+ [27][28] addresses the conceptual design of business processes in Web applications in two steps. In the first step a business process is modeled by means of one or more Web Transactions, a set of activities the user will be enabled to carry out by means of the Web application and that will permit him to execute the business process. In the second step each activity included in the defined Web transaction is associated with a navigation cluster and a navigation node which are the way an activity will be presented to the user.

Each Web Transaction defined by the first design phase is designed by means of two models: an Organization model and an Execution model. The Organization model is a customization of the UML class diagram, in which activities in which the transaction is are arranged to form a tree; the main activity is represented by the root of the tree and corresponds to the entire transaction, while activities and sub-activities are intermediate nodes and their leaves. The Execution model represents a Web Transaction from a dynamic point of view. It is a customization of the UML Activity Diagram [29] where activities and sub-activities are represented by states (ovals), and the execution flow between them is represented by state transition (arcs).

The second phase enriches the Navigation model of the Web application with the activity nodes and the activity clusters associated with each of the activities included in the Web transactions defined with the first design step. An activity cluster is a design concept introduced in UWAT+ to design the possible navigation between a node and its "adjacent" and the interaction between activity execution and content navigation. Similarly to OOHDM navigation nodes, activity nodes are used by UWAT+ to aggregate the portions of content that will be presented to the user when executing a given activity, but also to define which action elements will be included in the node.

Even from the above simplified description of the UWAT+ model, we can highlight the usage of separation of concerns in the design of a business process in Web application.

As an example while the Organization and the Execution model of a Web transaction are focused in modeling the business rules of the implementing business process by defining the component activities, their execution flow and the collaboration of the involved actors, the navigation model focuses on defining the aspects of a Web transaction related to the content navigation and provision.

3.2 Impact on Evolution

Separation of concerns adopted by UWAT+ design model helps evolution of the implemented Web application in the following maintenance phase of its life-cycle.

Suppose we wish to change the contents that have to be provided to user when executing one of the activities of the business process. We need to update the navigation model of the application and from it to derive the changes to be done to the implementation code of the application. The organization and execution model of the implemented process won't be affected, and thus the business logic of the application.

Suppose instead that some of the business rules of the implemented business process change. In this case we will probably need to update the organization and/or execution model of the Web transaction that models the business process in the Web application and derive from the changes applied to these models the portions of the application code to change. If no change is required to the navigation model, only code related to the implementation of the business rules will be affected.

The usage of UWAT+ to reengineer and evolve a business Web application by means of the reengineering and evolution of its Web transactions can be found in [30][31].

4 Concluding Remarks

Aspects, roles and views help separating design concerns during the design of a Web application and separating design concerns ease the evolution of Web applications during maintenance time.

In this paper we have examined how two well established design methodologies, OOHDM and UWAT+, aimed at designing Web applications the first and specifically tailored for addressing the design of business processes in Web applications the second, make use of separation of design concerns in their conceptions and how the resulting design approaches and models ease the evolution of the implemented applications.

References

1. Allert . H., Dolog, P., Nejdl, W., Siberski, and W., Steimann, F.: Role-oriented Models for Hypermedia Construction – Conceptual Modeling for the Semantic Web -. Technical Report, Univ. Hannover (2003)
2. Hubert Baumeister, Alexander Knapp, Nora Koch and Gefei Zhang. Modelling Adaptivity with Aspects. In 5th International Conference on Web Engineering (ICWE 2005), to appear.
3. Bäumer, D., Riehle, D., Siberski, W. and Wulf, M.: The Role Object Pattern. In Proceedings of Pattern Languages of Program Design (PloP) (1997) Available at: http://jerry.cs.uiuc.edu/~plop/plop97/Proceedings/riehle.pdf
4. Ceri, P., Fraternali, P., and Bongio, A.: Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. Computer Networks and ISDN Systems, 33(1-6), June (2000) 137-157
5. De Troyer, O.: Audience-driven web design", In Information modelling in the new millennium, Eds. Matt Rossi & Keng Siau, Publ. IDEA GroupPublishing (2001)
6. Fowler, M.: UML Distilled. Addison Wesley (1997)
7. Gamma, E., Helm, R., Johnson, and R., Vlissides, J.: Design Patterns: Elements of reusable Object-Oriented Software. Addison Wesley, Reading (1995)
8. Halpin, T.: UML Data Models from an ORM Perspective: Part Five, Journal of conceptual modeling, Issue 5 (Oct. 1998)

9. Koch, N., Kraus, A., Hennicker R.: The Authoring Process of UML-based Web Engineering Approach. In Proceedings of the 1st International Workshop on Web-Oriented Software Construction (IWWOST 02), Valencia, Spain (2001) 105-119
10. Koch, N. Kraus, A.: Towards a Common Metamodel for the Development of Web Applications. In 3rd International Conference on Web Engineering (ICWE 2003) Cueva, J., Gonzalez, B., Joyanes, L., Labra, J. and Paule, M. (Eds) LNCS 2722, ©Springer Verlag (July 2003) 497-506
11. Kristensen, B.B., Osterbye, K.: Roles, Conceptual Abstraction Theory and practical Language Issues. Theory and Practice of Object Systems, 2(3) (1996) 143-160
12. Open Hypermedia Systems Working Group home page: In http://www.csdl.tamu.edu/ohs/
13. Pernici, P.: Objects with Roles. Proceedings of the ACM-IEEE Conference on Office Information Systems (1990) 205-215
14. Reina, A.M., Torres, J.: Analyzing the Navigational Aspect, In Second German Workshop on Aspect-Oriented Software Development, University of Bonn, February 2002.
15. Riehle, D.: Role Model Based Framework Design and Integration. In Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98). ACM Press (1998) 117-131.
16. Trygve M. H. Reenskaug: Working with objects. The OOram Software Engineering Method. Manning/Prentice Hall (1996)
17. Rossi, G. Nanard, J., Nanard M., Koch N.:"Engineering Web Applications with Roles". Submitted Paper. Also LIRMM, University of Montpellier Technical Report, May 2004.
18. Rossi, G., Gordillo, S. and Schwabe, D.: Separation of Structural Concerns in Physical Hypermedia Models, In Proceedings of CAiSE 2005, Springer Verlag, LNCS 2005, forthcoming.
19. Schmid, H., Rossi, G.: Modeling and Designing Business Processes in Web Applications, In IEEE Internet Computing, February/March 2004.
20. Schwabe, D and Rossi, G.: An Object-Oriented Approach to Web-Based Application Design. Theory and Practice of Object Systems (TAPOS), Vol 4 (1998) 207-225
21. Schwabe, D. and Rossi, G.: Web Application Models are more than Conceptual Models. In Proceedings of the International Workshop on Conceptual Modelling and the Web (1999)
22. Sowa, J.: Conceptual Structures: Information Processing in Mind and Machine. Addison Wesley (1984)
23. Steimann, F.: On the Representation of Roles in Object-Oriented and Conceptual modeling. Data and Knowledge Engineering 35 (2000) 83-106
24. Steinmann F.: A radical revision of UML's Role Concept. In Proc. of the Unified Modeling Language Conference 2000, LNCS, Springer (2000) 194-209
25. UWA Consortium: Ubiquitous Web Applications, Proceedings of e2002 eBusiness and eWork Conference, Prague (Czech Republic), October 2002.
26. UWA Consortium: Deliverable D8: Transaction design., Online at www.uwaproject.org, 2001.
27. Distante, D.: Reengineering Legacy Applications and Web Transactions: An extended version of the UWA Transaction Design Model. Ph.D. Dissertation, University of Lecce, Italy. June 2004.
28. Distante, D. and Tilley, S.: "Conceptual Modeling of Web Application Transactions: Towards a Revised and Extended Version of the UWA Transaction Design Model" In Proceedings of the 11th International Multi-Media Modelling Conference (MMM

2005: Jan. 12-14, 2005; Melbourne, Australia). Los Alamitos, CA: IEEE Computer Society Press, 2005.
29. Object Management Group (OMG). Unified Language Modeling Specification (Version 2.0). Online at www.omg.org, 2004.
30. Tilley, S., Distante, D., Huang, S.: “Web Site Evolution via Transaction Reengineering.” To appear in Proceedings of the 6th International Workshop on Web Site Evolution (WSE 2004: Sept. 11, 2004; Chicago, IL). Los Alamitos, CA: IEEE Computer Society Press, 2004.
31. Tilley, S., Distante, D., and Huang, S.: “Design Recovery of Web Application Transactions.” In Advances in Software Evolution with UML and XML (Editor: Hongji Yang). Hershey, PA: Idea Group Publishing, May 2005.

Developing Navigational User Interfaces Using Business Processes

Qi Zhang[1], Rongchao Chen[2] and Ying Zou[3],
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
{3qz[1], 4rcc[2]} @qlink.queensu.ca, ying.zou@queensu.ca[3]

Abstract

Business users are often overwhelmed by the enormous functionality available in business applications, such as e-commerce applications, necessary to accomplish activities required by business processes. The business users are required to take continual training to use e-commerce applications, since the user interfaces of these e-commerce applications are frequently updated to reflect the continuous evolution of the underlying business processes. Business processes describe the functionality of e-commerce applications from the perspective of business users. In this position paper, we utilize the knowledge embedded in business processes and generate navigational user interface components that implement business tasks defined in business processes. We aim to provide contextual information and guide users to fulfill business tasks step by step in order to achieve business objectives.

1. Introduction

A business process is a sequence of tasks which need to be carried out to achieve business objectives for the organization. For example, a book purchasing process may consist of several tasks, such as selecting a book from the catalog, using a credit card for the payment, and printing out a receipt. A workflow provides specifications for describing business tasks, roles, data and resources involved in a business process. Business applications are designed to support business tasks specified in business processes. To achieve the escalating requirements from business users, business applications, such as e-commerce applications, have gradually evolved and provide sophisticated functional features in user interfaces. However, such rich features are often not obvious for users to navigate in the user interfaces (UIs). As a result, business users, especially novice business users may struggle in deciding where to start or where to go next after a result is received from a particular toolbar in the user interface. One of the reasons is that the user interfaces of business applications are often designed from the perspective of developers, but not from usability point of views of business users. Therefore, business users need to take continual training in order to be competent to perform business process activities using these business applications. These problems result in increase in operation cost and decrease in business performance. Therefore, a more systematic approach is required to design these business applications to ensure business users can carry out business activities in these applications more efficiently and effectively.

In this paper, we propose an approach for developing business process driven user interface. We leverage the knowledge in the business processes to improve the usability of user interfaces of business applications. We automatically generate navigational UI components which indicate the progress of a business process and automatically prompt the next task for the user to accomplish. Our aim is to guide the user to operate on an appropriate existing UI component that is developed to fulfill a business task. To integrate the generated navigational user interface components with existing UI components, we define the linkages between the two. Furthermore, we adapt contextual information embedded in business processes to dynamically provide the UI components relevant only to currently progressing business processes.

2. An Approach for Developing Business Process Driven User Interfaces

Business processes describe the functionality of a business application using a set of task specifications

in workflows. Moreover, a business process defines a possible processing order in which business users interact with the user interface components that fulfill business tasks. A business process also describes the contextual data and resources that assist the fulfillment of each business task. As a result, the knowledge in the business process can be leveraged and served as the initial design requirement for the user interfaces of the business application.

Typically, an UI component is represented as a dialog, an editor window or a view window in a business application. Each UI component may implement one or many tasks. This association is known as the binding between UI components and the tasks in a workflow [2][6]. As an example, a typical *Purchase order* process for a retail sales application is shown in figure 1. The *Enter customer information* task can be completed in an order editor window. The *Add product to order* task and the *Enter payment information* task can be fulfilled in the Select product dialog and payment dialog, and finally the *Submit order* task needs to be done in the same order editor. Essentially, the business user completes the business process by navigating to different UI components and performs tasks in each component. In our approach, we utilize this information to ease the uses of a user interface.

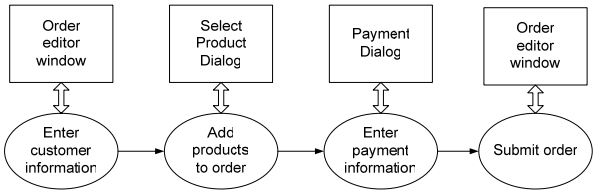


Figure 1: Binding tasks to user interface components for a Purchase order process

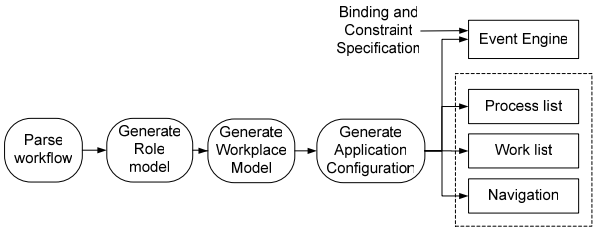


Figure 2: A framework for developing business process driven user interfaces

2.1 A Framework for Designing Business Process Driven User Interfaces

Our framework for developing business process driven user interfaces was detailed in our previous work [6]. The structure of our framework is shown in Figure 2. Our framework first parses workflows. It then generates the role model which describes tasks required for each business role (e.g., sales representative) to perform. The role model is further transformed into a workplace model, which defines the basic layout and content of the user interface components. In our framework, we automatically generate three UI components that facilitate users to access the tasks, including *Process List*, *Work list* and the *Navigation.* The *process list* allows the user to instantiate a workflow instance by selecting a process name in the list. The *Work list* is used to show the tasks that need to be completed by the user. The *Navigation* is used to display the flow structure as well the status of the workflow instance. Finally, the workplace model is converted into an application configuration, which is used to configure the layout and content of UI complements implemented by a particular UI technologies, such as the Web portals [8]and Eclipse Rich client platform [9]. The configuration is processed by the *event engine* to configure navigational UI components, including the *Process List*, *Work list* and the *Navigation* at run time. Configurations contain bindings and constraint specifications, which allows for integrating the navigational UI components with the existing business application. By incorporating business process flow information in the user interface, we provide a user-guidance environment for user to perform daily work. Detailed operations of each component are mentioned in the next section.

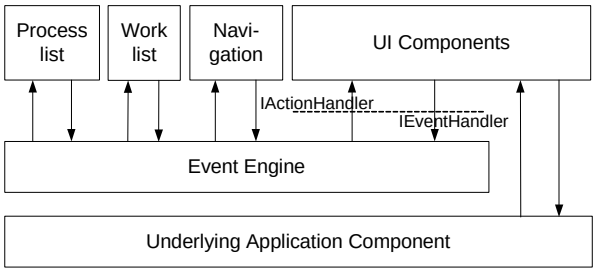


Figure 3: Architecture of Workflow-driven Business Application

2.2 Architecture for Event Engine

The architecture of our business process driven user interface for business applications is shown in figure 3. The *Process list*, the *Work list* and the *Navigation* components communicate with the event engine to obtain the run time workflow instance information. To solve the consistency issue between the user interfaces and its underlying business processes, we created a binding between the two, which includes an

event interface, known as *IEventHandler*, and an action interface, known as *IEventHandler*, as illustrated in Figure 4. The *IEventHandler* is used by the UI components to send events to the *event engine*. For example, a UI component can send an UI event that indicates the start or end of a task. The *IActionHandler* interface is used by the *event engine* to drive the UI components. The current actions supported include open, close, highlight and de-highlight UI components. In a typical execution sequence, when a task needs to be executed, the *event engine* automatically opens (or shows) the UI component to the user. During an instance switch, the event engine de-highlights the UI components bound to a UI component, and highlights the UI components bound to the new workflow instance. However, in certain cases, it is not desirable for event engine to automatically open the UI component, since the user may want to verify the result of the previous step. In this case, we allow the user to open the UI component manually.

When the user is working in one UI component, it sends events to indicate the start and end of the tasks. In addition, tasks can be aborted. For example, the user can select the Cancel button in a dialog to undo the changes in the dialog. In this case, a terminate event is also sent by the UI component. Further, some task may require no user interaction. For example UI component may provide a default value so the user does not need to perform any actions in the UI. Even more, some tasks can be done repeatedly. An example is the *Add products to order* task shown in figure 1. This task can be carried out multiple times, depending on the customer's decision. Moreover, some tasks are also restrained by additional constraints, such as application data and the execution sequence. These properties and constraints are captured in a constraint configuration. These constraints can be provided by the software developers based on their design of the business application.

```
public interface IActionHandler {
  public void OpenUIComp(String ApplicationName,
         Object data);
  public void HighlightUIComp (Object application);
  public void DehighlightUIComp (Object application);
  public void CloseUIComp (Object application);
}

public interface IEventHandler {
  public void fireWorkflowEvent(String eventName,
         Object[] data);
}
```

Figure 4: Definitions of the *IActionHander* and the *IEventHandler* interface

The advantage of our approach is that it supports multiple workflow instances. For example, two *Purchase order* process can run at the same time, in two different order editors. The *event engine* distinguishes run-time workflow instances based a set of unique identifiers. These identifiers could be the UI component itself, or the data being processed by the UI components. When UI event occurs, the event also sends the unique identifiers along with the event (represented by `Object[] data` in figure 4). Given the uniqueness of the identifiers, the *event engine* always knows which workflow instance is being processed by the user.

```
<Configuration>
<ProcessBinding ProcessName="Purchase Order">
  <TaskBinding TaskName="Enter customer information"
         Optional="false" Manual="false">
<Application Name="OrderEditor" Invoke="true"
         Close="false" />
  </TaskBinding>
  <TaskBinding TaskName="Add products to order"
         Optional="false" Manual="false">
    <Application Name="SelectProductDialog" Invoke="false"
         Close="false" />
  </TaskBinding>
  <TaskBinding TaskName="Enter payment information"
         Optional="false" Manual="false">
    <Application Name="PaymentDialog" Invoke="false"
         Close="false" />
  </TaskBinding>
  <TaskBinding TaskName="Submit order" Optional="false"
         Manual="false">
    <Application Name="OrderEditor" Invoke="false"
         Close="false" />
  </TaskBinding>
</ProcessBinding>
<TaskConstraint TaskName="Enter customer information">
  <TaskPrecondition>
    <EventConstraint
         EventName="Start Enter customer information" />
  </TaskPrecondition>
  <TaskPostcondition>
    <EventConstraint
         EventName="End Enter customer information" />
  </TaskPostcondition>
  <TaskCancelcondition>
    <EventConstraint EventName="Order editor closed" />
  </TaskCancelcondition>
</TaskConstraint>
    ...
</TaskConstraint>
<Configuration>
```

Figure 5: Binding and constraint specification for the Purchase order process

Another advantage of our framework is that since the event engine has the knowledge of the progress of each workflow instance, the event engine can provide support in the business application by displaying related information or tools related to the current task. When the task starts, the relevant information and tools automatically appears. When the task ends, these UI components are removed. Furthermore, when the user switch instance in the workplace, the supporting tools are automatically displayed to or hidden from the user, based on the context of the current progressing workflow instance.

2.3 Binding and Constraint Format

In this section we elaborate more on the format of our binding and constraint specification. The binding specification is used to specify the linkage between UI component and the tasks in the business process, and to integrate navigational UI components with existing UI components. The constraint specification describes each event produced by the UI components and the mapping between events and the start and end condition of tasks. This provides flexibility for the developers to modify the source code to generate events in the UI components.

As an example, the binding and constraint specification for our example process in figure 1 is shown in figure 5. The first part is the binding specification that defines the binding between UI component and the workflow task. It also contains the properties of the tasks. A task can be manual, optional or repeatable. A manual task is a task which is performed without interacting with the business application. In this case, the task is not bound to any UI components. An optional task is a task which may require no user interaction. A repeatable task is a task that can be done repeatedly. The *Invoke* and *Close* property is used to indicate whether the UI component needs to be opened or closed automatically once the task needs to be executed or completed. The second section is the constraint specification. Each task in the process is associated with pre-conditions, post-conditions and cancel-conditions. Each condition specifies the start, end and abort condition of the task. Each condition is composed of constraints, such as event constraint, data constraint and workflow structural constraints. These constraints allow the event engine to interpret events and update the state of workflow instances correctly.

3. Case Studies

To further examine our process driven user interface framework, we developed a prototype tool in Eclipse, which is an open extensible platform and Integrated Development Environment (IDE) for Java programming. We integrate our framework with a call center application with 100K LOC. This system provides full functionality for a sales representative to interact with customers and sales systems, however, with drawbacks and limitations in terms of usability and consistency with its underlying business processes. For example, novice users will be overwhelmed by its numerous functions provided through its menu items and may get lost during operation since some proceeding buttons are hidden deep inside. The e-commerce application that we used in our research inherits the Eclipse presentation style, which includes its workbench, views, editors and menu items, and is known as Eclipse Rich Client Platform (RCP). Our prototype tool is integrated with the call center application as an Eclipse plug-in. The screenshot of the prototype system is illustrated in Figure 6.

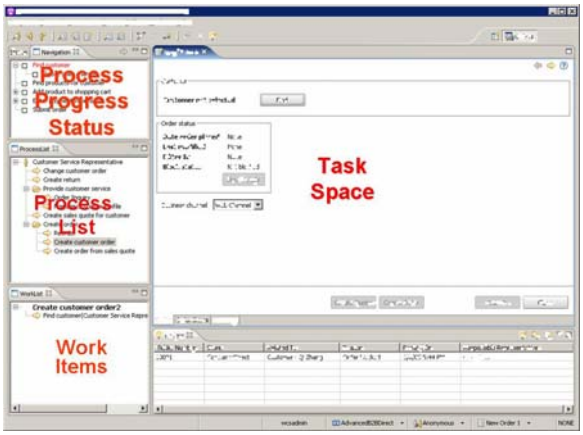


Figure 6: Prototype Process Driven User Interface

Within the original call center sales application, events were inserted into the user interface components classes that are tightly bound to business workflow tasks, such as the creation of an order editor instance which is related to the create order task in business process. There are approximately 30 events in total and they are handled by the event engine within our plug-in project and corresponding actions are performed from the business task that is bound to the event through the pre-defined action interface.

Bindings between business processes and application user interface are achieved by using a configuration file. In order to maintain the integrity fo the original design of UI components, we decided to set the level of granularity in the binding to user interface components such as editors, views and dialogs as comparison with other research [2] that binds tasks to lower levels such as widgets. The template for this binding configuration is automatically generated by parsing through business process and workflow graphs. The developers only need to fill in the user interface component configuration when using it.

Our prototype tool enhances the application's usability and displays underlying business processes information. However, our current implementation still has limitations. Since the business processes are not defined based on the user interface, there is always gap between the two. For instance, multiple business processes can be implemented on one user interface component to save operation time. Thus, the mapping

between business tasks and user interface components can be difficult to identify in some cases.

4. Conclusion

In this paper we proposed an approach for developing navigational user interfaces using business Processes. We aim to provide guidance to facilitate users to fulfill business tasks. In the future, we would like to further refine our approach. We plan to conduct usability studies to test the effectiveness of our approach.

Acknowledgement

This research is sponsored by IBM Canada, Centers for Advanced Studies (CAS), and National Sciences and Engineering Research Council (NSERC).

We would like to thank Jen Hawkins, Bhadri Madapusi, Tack Tong and Ross McKegney for their valuable contribution to this work.

References

[1] "From business reengineering to business process change management: a longitudinal study of trends and practices", *IEEE Transactions on Engineering Management*, Vol. 46 No. 1, pp. 36-46.

[2] Sliski, T.J., Billmers, M.B., Clarke, L.A. and Osterweil, L.J. "An Architecture for Flexible, Evolvable Process-Driven User Guidance Environments" *Proceedings of the Joint 8th European Software Engineering Conference (ESEC 2001) and the 9th ACM Sigsoft Symposium on the Foundations of Software Engineering (FSE 9)*, September 2001, Vienna Austria, pp. 33-43.

[3] Ying Zou, Terence Lau, Kostas Kontogiannis, Tack Tong, Ross Mckegney, "Model Driven Business Process Recovery", *in the Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE)*, Delft, The Netherlands, Nov. 2004.

[4] Becker J., zur Muehlen M., Gille M "Workflow Application Architectures: Classification and Characteristics of Workflow-Based Information Systems", *Workflow Handbook 2002*, Future Strategies, Lighthouse Point, 2002.

[5] D.-A. Manolescu and R. E. Johnson. A micro workflow framework for compositional object-oriented software development. OOPSLA'99 Workshop on the implementation and Application of Object-Oriented Workflow Management Systems II, Nov. 1999.

[6] Qi Zhang, Ying Zou, Tack Tong, Ross MacKegney and Jen Hawkins, "Automated Workplace Design and Reconfiguration for Evolving Business Processes", *to appear in the 15th Centre for Advanced Studies Conference (CASCON)*, Toronto, Canada, November 2005, pp. 262-277.

[7] "Business Process Execution Language for Web Services version 1.1". ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

[8] Corporate Portals Empowered with XML and Web Services, Butterworth-Heinemann Newton, MA, USA, 2002

[9] Eclipse Rich Client Platform. http://www.eclipse.org/rcp/

Workshop 2.2:
Workshop on Empirical Studies in Reverse Engineering

Session: Tools and Techniques

Columbus: A Reverse Engineering Approach

Árpád Beszédes, Rudolf Ferenc and Tibor Gyimóthy
University of Szeged, Department of Software Engineering
and
FrontEndART Software Ltd.
{beszedes|ferenc|gyimi}@inf.u-szeged.hu

Abstract

In this paper we present our approach to several common problems in reverse engineering that are built around the Columbus framework. Columbus defines several fundamental building blocks for the use in reverse engineering processes, and as such it can be an important player in the studies conducted at the workshop for Empirical Studies in Reverse Engineering. The Columbus framework proved its usefulness in the field through a number of research projects (also by independent researchers) and several industrial applications. Columbus may contribute as (1) a flexible, easily extensible tool architecture, (2) a data exchange model (C/C++ schema) and (3) as a source code analysis process.

1. Introduction

The role of reverse engineering in software evolution is evidently important. However, in order to introduce this discipline into the software engineering process, some fundamental building blocks are needed. Such are the availability of tools, data exchange settlements and a reverse engineering process.

We demonstrate the reverse engineering approach that builds upon the *Columbus* framework [4, 5, 6], which addresses these essential elements, and as such can be an important player in a study conducted to assess the existing approaches and define a general framework for empirical studies. The Columbus technology is by now recognized by the academia, in fact it is one of the reference architectures in the field. In addition to the numerous examples of its usage by research groups, it also plays an important role in a number of industrial projects.

Columbus deals with the issues mentioned above in the following way. First, it includes an extensible reverse engineering tool, currently with a source code analysis module for the C/C++ language. Second, it defines a schema for representing C++ source code as an exchange model among other tools [3, 6, 12]. This schema is also one of the reference representations according to state of the art. Finally, Columbus contributes to the reverse engineering process in the following manner. Although it does not cover all potential aspects of such a process, it deals with some of the most critical issues such as project set-up and fact extraction. We emphasize the importance of the former, since in many situations collecting the artifacts to be analyzed is not trivial, and being a fundamental activity, it affects all remaining parts of the process. The most important aspect that needs special attention of the latter issue, fact extraction, is that it needs special design consideration regarding code analysis, with respect to traditional methods such as those used by compilers.

We overview fact extraction using Columbus in the second section, while data exchange is discussed in Section 3. Section 4 deals with representing the extracted data in another form and providing input to other tools, and in the final section we summarize our contributions to the workshop.

2. Fact extraction with Columbus

To comprehend a software system we need to know many different things about it. We refer to this information as *fact*s about the source code. *Fact extraction* is an automatized process during which the subject system is analyzed with analyzer tools to identify the source code's various characteristics and their interrelationships, and to create abstract representations of the extracted information. The form of these representations is prescribed by *schemas*, which are descriptions of the form of the data in terms of a set of entities with attributes and relationships. A *schema instance* is an embodiment of the schema which models a concrete software system. To make the results of fact extraction widely usable, we further process the schema instances to take various new formats.

Columbus is a reverse engineering framework developed in cooperation between the University of Szeged, the Nokia

Research Center and FrontEndART [9]. The main motivation behind developing this framework was to create a toolset which supports fact extraction and provides a common interface for reverse engineering tasks in general. The graphical user interface of the framework is called *Columbus REE* (Reverse Engineering Environment). Further tools are also incorporated into the framework (mostly command line), which actually do the C++-specific tasks, like analyzing the source code and further processing the results.

The extraction process within the Columbus framework is outlined in [4]. The process is very similar to the traditional compilation process. It consists of five consecutive steps (see Figure 1) where each step uses the results of the previous one.

The steps of the process are the following:

1. Acquiring project/configuration information
2. Analysis of the source – creation of schema instances
3. Linking of schema instances
4. Filtering the schema instances
5. Processing the schema instances

These steps may be performed in different ways: using the visual user interface of the Columbus REE, using the compiler wrapper toolset (see below), or using only the command-line programs by themselves. An important advantage of the presented steps is that they can be performed incrementally, that is, if the partial results of certain steps are available and the input of the step has not been altered, these results need not be regenerated.

Acquiring project/configuration information is indispensable to carry out the extraction process. The source code of a software system is usually logically split into a number of files and these files are arranged into folders and subfolders. Furthermore, different preprocessing configurations can apply to them. The information on how these files are related to each other and what setting apply to them are usually stored either in *makefile*s (in the case of building software with the *make* tool), or in different *project files* (in the case of using different *IDE*-s – Integrated Development Environments).

The Columbus technology employs a so-called *compiler wrapping* technique for using makefile information and two different approaches for handling IDE project files: IDE integration and project file import.

Compiler wrapping. Makefiles can contain not only the references to files to be compiled and their settings but can also contain various commands, like invoking external tools. These powerful possibilities are bad news for reverse engineers, because every action in the makefile must be somehow simulated in the reverse engineering tool. This can be extremely hard or even impossible in some cases. We approached this problem from the other end and solved it by "wrapping" the compiler. This means that we temporarily hide the original compiler, and this way if the original compiler should be invoked our wrapper program will start instead of it, which executes first the original compiler, and second, it invokes our analyzer tools as well. These are invoked with the appropriate parameters in the same environment to build up the required schema instances. This way all we have to do is to build a software system as usual (but with the wrapper switched on).

IDE integration. In this case our tool appears as a new toolbar within the IDE and its operation is very similar to the usual build process. The active project is analyzed and the output can be transformed into any format supported by the Columbus framework. (Currently Microsoft Visual Studio 6.0 and .NET are supported.)

Project file import. The Columbus REE is able to parse Microsoft Visual C++ 6.0 and .NET project files and to import all relevant information from them to be able to analyze the project. All Columbus REE features can be used in this case.

Manual setup. Besides these possibilities the project can be built up by hand also in the Columbus REE (for instance if no project information is available at all). A so-called *Project Setup Wizard* is available to help in this task.

The Columbus environment currently contains a C/C++ source code analysis front end, which is constituted of a specially developed preprocessor and language analyzer. The analyzers were designed especially to meet the requirements of source code analysis in the scope of reverse engineering. The employed technology makes possible, for example, parsing incomplete code, source-complete detailed analysis, and the handling of language dialects.

3. Data exchange

Successful data exchange is crucial among reverse engineering tools. This requires a common *format*, which is applicable in various reverse engineering tools such as front ends and metrics tools. A standard schema must be found. We described our approach to this important topic in [3, 6, 8], which is since then known as the *Columbus Schema for C++*, and it describes the C++ language details. An extension to the C++ schema is the schema for capturing the *preprocessor* related facts as described in [12].

The Columbus schemas capture the C++ and preprocessor languages at low detail (AST) and also contains higher-level elements (e. g. semantics of types). The description of the schemas is given using UML Class Diagrams, which permits their simple implementation and easy physical representation (e. g. using GXL). Their modularity provides additional flexibility for any future extension/modification. The implementation of the classes belonging to the schemas provides an Application Programming Interface for access-

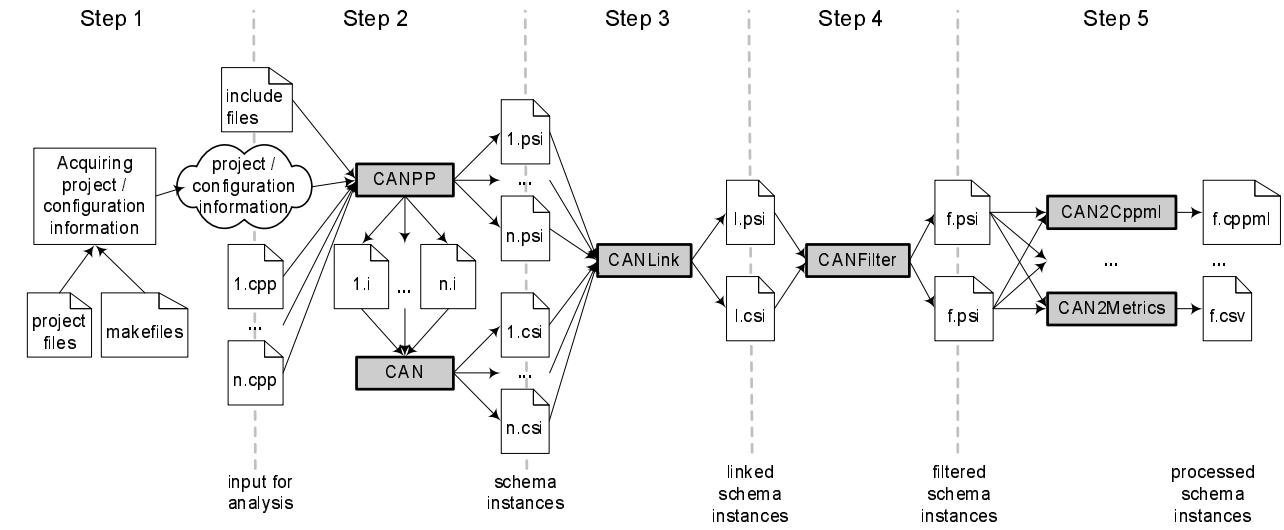


Figure 1. The fact extraction process

ing the internal representation, which is used both by the Columbus framework itself and independent developers as well.

Apart from defining the content of the data to be represented and potentially exchanged with other reverse engineering tools, it is also necessary to settle down common physical formats. To this end, the Columbus schemas can be represented in various external formats, including the increasingly popular GXL format. The external formats are overviewed in the following section.

4. Presentation of data and tool interoperability

Because different re- and reverse engineering tools use different schemas for representing their data, the schema instances must be further processed to achieve tool interoperability. The processing may consist of transforming the schema instance into another format and/or applying further computations on it. Currently the following are included in the Columbus REE:

PPML and CPPML. This transformation permits the creation of XML documents (called *PPML* – Preprocessor Markup Language and *CPPML* – C++ Markup Language) that have structures based on the corresponding Columbus schemas. The exported documents conform to their Document Type Definitions, as described on FrontEndART's homepage [9].

GXL. With this transformation GXL representations can be created from the extracted information. *GXL* (Graph eXchange Language) [10] is an XML-based graph-description format. Since the Columbus schemas basically define graphs, this format is suitable for representing them in a convenient way. The call graph of the analyzed system can be also created in GXL form.

UML XMI. This processing allows the creation of standard UML XMI documents from the Columbus Schema for C++. The XMI document contains the class diagram of the analyzed project which can be further processed with XMI enabled tools (like IBM Rational Rose, Borland Together ControlCenter, etc.).

Famix XMI. With this processing a *Famix* [2] XMI representation of the extracted information can be created. This format can be utilized in the *CodeCrawler* tool for visualization and metric calculations.

RSF. Three transformations are available for creating *rigi* RSF [11] documents: (1) a graph based on the Columbus Schema for C++, (2) a call-graph and (3) a UML class diagram-like graph. All of these use different rigi domains which can be created with Columbus as well.

HTML documentation. This processing can be used to create a hypertext documentation of the extracted project in *HTML* form. The generated documentation presents the project in a browsable and user-friendly fashion. All the necessary information is presented about the classes and other elements in a structured way. Three types of browser frames are also supplied, with which a project can be easily navigated. These present the classes using (1) their names in alphabetical order, (2) the scoping structure and (3) the inheritance relationship.

Apart from these, so to say, simple transformations, we have been experimenting with other kinds of processings of the extracted fact representations. These include the calculation of *object oriented metrics* [7], recognition of *design patterns* [1] and *bad smells* (for refactoring purposes) in the code.

Furthermore, we have developed a special code auditing tool based on Columbus – called *SourceAudit* – which is able to investigate source code and check it against rules that describe the preferred properties of the code. These rules mostly involve issues related to coding style, but in some cases they extend the warning reporting capabilities of the compiler. The checked rules are organized into rule packages and the tool can be freely extended with new packages. The tool can be used in command line and integrated with popular IDE-s (e. g. Microsoft Visual Studio and Borland C++Builder).

5. Relevance to the Workshop

We believe that the Columbus reverse engineering technology may serve as one of the fundamental approaches to be investigated under the scope of the workshop for Empirical Studies in Reverse Engineering. It may contribute as a tool, a data exchange model and as a source code analysis process.

The Columbus Reverse Engineering Environment employs a highly flexible architecture that is essential for tool reuse in various environments where reverse engineering is needed. Its C/C++ source code analysis front end includes the necessary analysis technologies that are specific to the purpose of reverse engineering.

The extracted facts about the source code are stored and further manipulated according to a language schema that, together with common formats like GXL, provides a basis for successful data exchange among reverse engineering tools. Columbus supports a number of external formats that are most commonly used by the reverse engineering community, and this interoperability helped being utilized in several research and industrial projects.

One of the most important assets of Columbus is that it performs the analysis tasks according to a proven source code analysis process. It includes several ways of setting up the reverse engineering projects, because we believe that this is one of the most critical issues in successful code analysis. The compiler wrapping technology proved its simplicity and usefulness in a number of successfully performed analyses of real size software systems, like the open source Mozilla and StarOffice systems.

References

[1] Z. Balanyi and R. Ferenc. Mining Design Patterns from C++ Source Code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society, Sept. 2003.

[2] S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization. In *Proceedings of WCRE'99*, 1999.

[3] R. Ferenc and Á. Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, Mar. 2002.

[4] R. Ferenc, Á. Beszédes, and T. Gyimóthy. Extracting Facts with Columbus from C++ Code. In *Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 4–8. IEEE Computer Society, Mar. 2004.

[5] R. Ferenc, Á. Beszédes, and T. Gyimóthy. *Tools for Software Maintenance and Reengineering*, chapter Extracting Facts with Columbus from C++ Code, pages 16–31. Franco Angeli Milano, 2004.

[6] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, Oct. 2002.

[7] R. Ferenc, I. Siket, and T. Gyimóthy. Extracting Facts from Open Source Software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, Sept. 2004.

[8] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, and T. Gyimóthy. Towards a Standard Schema for C/C++. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 49–58. IEEE Computer Society, Oct. 2001.

[9] Homepage of FrontEndART Software Ltd. `http://www.frontendart.com`.

[10] R. Holt, A. Winter, and A. Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of WCRE'00*, pages 162–171, Nov. 2000.

[11] H. A. Müller, K. Wong, and S. R. Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Proceedings of ACFAS*, 1994.

[12] L. Vidács, Á. Beszédes, and R. Ferenc. Columbus Schema for C/C++ Preprocessing. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 75–84. IEEE Computer Society, Mar. 2004.

Software evolution the need for empirical evidence

[Position Paper]

Giuliano Antoniol, Yann-Gaël Guéhéneuc, Ettore Merlo, and Houari Sahraoui
Ecole Polytechnique and University of Montreal
Montreal, Quebec, Canada
antoniol@ieee.org, ettore.merlo@polymtl.ca, {guehene,sahraouh}@iro.umontreal.ca

An intrinsic property of software is its malleability, the fact that it may change and evolve. Software evolution is costly, because software systems tend to be highly complex and large. They are highly human-intensive and risky, because unplanned and undisciplined changes in any software system of realistic size risk degrading software quality and may produce unwanted and unexpected side effects. As a software system is enhanced, modified, and adapted to new requirements, its code becomes increasingly complex, often drifting away from its original design. The current state-of-the-art in software evolution offers only short-term solutions to software change and evolution focused on software maintenance and defect repair, in which only the source code evolves, while the architecture, design, and—more generally—the documentation are not updated.

The recent change towards a global economy poses new challenges with respect to software evolution. Software systems represent valuable assets to be preserved over time. Change and evolution management policies and strategies involve at least two different levels of granularity:

- First, the **macro level** at which a company manages its IT portfolio and makes macro-economic decisions. For instance, should a company wish to replace an existing IT component with a new software system, it must first conduct a feasibility assessment. The variables involved include the company's financial status, anticipated market trends, the costs of acquiring a new software system and training employees, total cost of ownership of the new software, and the cost of data migration. Other variables driving the decision have to be considered as well: the risks associated with possible failure in data migration or with the software vendor's financial status and liability.
- Second, the **micro level** refers to project-level company decisions. Once the decision to evolve the proprietary software has been made, the software change and evolution process must be implemented in facts. Implementation in turn calls for decisions such as partially or entirely outsourcing the maintenance and evolution process, defining the staffing level and evolution plan, reshaping the workforce, organizing the work packages, and defining schedules.

The already complex situation is made even more challenging by a new industry trend. Traditionally, the choice was either to *make* or to *buy*. However, the new market created by the service-oriented paradigm offers a new opportunity: Instead of buying or making, it is now possible to *rent* a service.

To make the situation even more complex, the cost of labor in emerging[1] countries favor outsourcing and distributed software evolution. For example, in 2003, the Russian offshore programming industry earned total revenues of 546 millions $US, according to figures compiled by CNews Analytics and Fort-Ross, an association of Russian software companies. The same report projected growth rates of 30% to 40% for the coming few years, meaning that the industry could cross the 1 billion $US by 2006. Overall, according to the analysis firm Datamonitor, the IT outsourcing market grew by 37% to 163 billions $US in 2004.

Distribution enables hiring the best experts without regards for geographical distances, thus reducing costs. Outsourcing rates tend to vary by the strength of the home economy and its degree of maturity of its IT industry. The cheapest alternative today is China, followed by Eastern European countries. India has a more mature IT industry, which makes outsourcing there more costly. However, although distribution allows emerging countries to attract investments, it also tends to draw resources and job opportunities from developed economies to relocation in third world countries.

[1] By "emerging countries", we mean countries in which the IT industry is becoming mature and a major source of incomes for the countries.

As a research community, we share the responsibility to support economic growth, employment, and training of highly qualified personnel. Thus, any action is important that may lead to identify approaches, methodologies, and tools, capable of enriching our current body of knowledge and of contributing to understanding the mechanisms in distributed software evolution. Indeed, labor cost and conditions are a major social issue worldwide. We are responsible of the effects and influences of our research results on these labor cost and conditions. In particular, we are responsible for devising approaches, methodologies, and tools to improve the general economy worldwide, such that outsourcing is beneficial to all, considering the difference of labor cost between developed and emerging countries (about an order of magnitude). This is a matter of scientific ethics.

Distributed software evolution needs convincing evidences to support the adoption of cost-effective and ethical solutions, as well as solutions that prove to be beneficial worldwide in the long term. For example, regardless of the adopted paradigm, technology, tools, there exists a very limited body of empirical results to help deciding what part of the software evolution process should be outsourced, where and when it should be outsourced, and what are the short-, mid-, and long-term effects of such a decision. Other engineering disciplines developed throughout time a body of knowledge to choose methods and to predict results effectively. By mean of standardization, subcontract standardization, and the adoption of components, industry in general increases productivity and decreases costs substantially. In software engineering, we are still far from the maturity of other engineering fields. Researchers and practitioners struggle to provide guidelines, evidences, and to meet predicted costs, schedules, and quality levels. Clearly, there are commonalities and differences with general industry, and general industry approaches may not apply in software engineering. (In particular, software is immaterial: There is easy means neither to verify where it was developed, maintained, evolved, nor to verify if it complies with standards, regulations, labor conditions.) However, as a research community, we must strive in developing approaches, methodologies, and tools, to meet practitioners' and societies' need alike.

In field validation, blind and double blind studies are mandatory to pharmaceutical industry prior a drug enter the market; in the same area replicated studies and studies disproving prior beliefs are as relevant as studies suggesting a beneficial application of a certain drug to a given disease. Standard ways to design and to carry out drug experimentations have been defined and are used. A solid scientific basis and legislation attempt to prevent misconduct experiments, flaw experiment design, wrong assumption and conclusion. We believe that in the software engineering area much as for pharmaceutical companies there is a need for a more solid scientific basis, a standardized way to perform experiments, more replicated studies, and rules for accepting replication studies in conferences and journals.

We identify several research questions that ought to be address by our research community regarding distributed software evolution:

- How could levels of productivity be increased? What approaches, methodologies, and tools use?
- Which parts of the distributed software evolution can be de-localized? What are the conditions to verify, the constraints to check, the controls to perform?

More generally, we believe that our research community must strive to answer the following questions:

- Is maintenance a fatality? How is maintenance performed? How is maintenance validated?
- How to compare the results of empirical software engineering studies? What approaches, methodologies, and tools should be applied?
- How to validate pieces of software (as for drugs)? Is it possible, desirable?
- How to perform and to publish replication studies?

How Software Repositories can Help in Resolving a New Change Request

Gerardo Canfora, Luigi Cerulo
RCOST — Research Centre on Software Technology
Department of Engineering - University of Sannio
Viale Traiano - 82100 Benevento, Italy
{canfora, lcerulo}@unisannio.it

Abstract

In open source development, software evolution tasks are usually managed with a bug tracker system, such as Bugzilla [1], and a versioning system, such as CVS [2]. This provides for a huge amount of historical data regarding bug resolutions and new enhancement feature implementations.

We discuss how software repositories can help developers in managing a new change request, either a bug or an enhancement feature. The hypothesis is that data stored in software repositories are a good descriptor on how past change requests have been resolved. Textual descriptions of fixed change requests stored in software repositories, both Bugzilla and CVS, are used to index developers and source files as documents in an information retrieval system. For a new change request, such indexes can be useful to identify the most appropriate developers to resolve it, or to predict the set of impacted source files.

1. Introduction

CVS and Bugzilla are two tools for configuration management used with success by the open source community for sharing knowledge during the software development process. The quality of data, in particular free text, such as bug comments, bug descriptions, and feature proposal definitions, is a critical need in an environment in which no people meetings, no phone calls, and no coffee break discussions are possible. This leads to consider such software repositories interesting data sources, useful for developing text mining techniques to assist project managers and developers in their maintenance activities.

Data contained in software repositories have generated new opportunities in various directions such as change propagation [11], fault analysis [7], software complexity [5], software reuse [9], and social networks [8]. In this paper we discuss how free text contained in software repositories can help developers in managing a new change request (CR), either a bug or an enhancement feature, and what are the new opportunities of mining free text software repositories.

2. Free text in software repositories

The resolution of a new CR, in many open source projects tracked by Bugzilla and CVS, usually follows a very simple workflow. A reporter propose a new CR that can be a bug he/she discovered or an enhancement feature he/she likes to suggest. The CR is stored in the new CR database, after a validation performed by the maintainer of the project to confirm it exists. A volunteer developer can pickup a new CR on the basis, for example, of its familiarity with the topic. An *assigned to* relationship links the developer with the CR. The resolution of the CR evolves becoming a fixed CR with a commit, in the CVS database, of the source code changes that resolves the CR (*impact* relationship). This relation cannot be recovered directly from Bugzilla database but needs to be derived by considering some heuristics [6]. Usually developers keep track of the files impacted by a CR by inserting in a CVS commit comment the id number of the CR tracked by Bugzilla. A *resolved by* relationship links the developer, author of the resolution change, and the fixed CR.

This process involves many information both structured and not structured, e.g. composed of free text. An example is shown in the following excerpts of a group of file revisions and the corresponding Bugzilla CR in XML format. Both have been retrieved from the software repositories of the Kpdf system.

```
revision 1.42 date: 2004/11/30 23:27:02;
author: aacid; state: Exp; lines: +10 -4
Watch files as kghostview does (use the very same code
:-D) Will be available in KDE 3.4
FEATURE: 94016
----------------------------
```

```
revision 1.41 date: 2004/11/01 21:56:14;
author: aacid; state: Exp; lines: +3 -0
Bring the document to the page it was when the session
was closed on session restore
FEATURE: 92503
----------------------------
revision 1.40 date: 2004/10/17 18:41:08;
author: aacid; state: Exp; lines: +1 -1
QString s -> const QString &s
...
```

A file revision is composed by a set of fields: *revision*, is a number that increases when new changes are committed by the developer; *date*, is the date and time of check-in; *author*, is an identificator of the person who did the check-in; *state* assumes one of the following values: 'exp' means experimental and 'dead' means that the file has been removed; *lines*, the number of lines added and deleted with respect to the previous version of the file; and a final block of free text that contains informal data entered by the developer during the check-in process.

```
<bug>
<bug-id>94016</bug-id>
<creation-ts>2004-11-27 07:33 PST</creation-ts>
<short-desc>Watch file option in kpdf</short-desc>
<product>Kpdf</product>
<component>General</component>
<bug-status>RESOLVED</bug-status>
<reporter>kde@foxcub.org</reporter>
<assigned-to>tsdgeos@terra.es</assigned-to>
<long-desc>
<who>kde@foxcub.org</who>
<bug-when>2004-11-27 07:33 PST</bug-when>
<thetext>KPDF would become a tremendously useful
application if it had an option to watch a file [...]
This is particular useful when making presentations in
LaTeX [...] </thetext>
</long-desc>
...
</bug>
```

A CR is enclosed in a *bug* tag, which contains: *bug-id*, a unique identifier assigned by Bugzilla; *creation-ts*, the date and time of CR creation; *short-desc*, a short description; *product*, the product name; *component*, the component of the system; *reporter*, who has submitted the CR; *assigned-to*, who was assigned the CR for resolution; and *long-desc*, a structure comprising a long description of the CR, *thetext*, who submitted it, *who*, and when, *bug-when*.

How this text can be used to support change request management is briefly described in the next section with two approaches that exploits well known information retrieval techniques.

	Kcalc	Kpdf	Kspread	Firefox
top 1 precision	78%	45%	39%	36%
top 30 recall	98%	85%	79%	67%

Table 1. Performance of source file prediction

3. What can be indexed in software repositories?

Given a set of text documents and a query describing a user information need as free text, the information retrieval problem is to retrieve all documents relevant to the user [10].

The application of information retrieval techniques on software repositories needs to define the terms *document* and *query* in the large. Documents can be anything which can be represented as free text. In the following two examples we consider, respectively, documents as source files and developers.

3.1. Index of source files

Indexing source files with free text contained in CRs are useful for the impact analysis problem [3]. A source file SF_i is represented as: $SF_i = \{CR^j_{sd}, CR^j_{ld}\}_{j \in impactOn(SF_i)} + \{R^k_{notes}\}_{k \in revisionOf(SF_i)}$, where CR^j_{sd} and CR^j_{ld} are respectively the short and the long description of the CRs that impact the source file and R^k_{notes} is the revision notes of the source file. The hypothesis is that the textual data carried by the bug tracking system during the resolution of a change request is a good descriptor of the impacted files to be considered in the impact analysis of future similar change requests.

Table 1 reports a summary of the results obtained in the validation we have performed in [3]. Top 1 precision, for example in Kcalc, means that the source file retrieved in the first position by an information retrieval algorithm is correct in 78% of cases. Instead, top 30 recall, in the same example, means that the source files retrieved in the first 30 positions by an information retrieval algorithm covers in average the 98% of correct files.

3.2. Index of developers

Indexing developers gives the opportunity for the project manager to identify the set of best candidate developers to resolve a new CR [4]. The hypothesis is that the best candidate developers are those that have resolved similar CRs in the past. A Developer D_i is represented as: $D_i = \{CR^j_{sd}, CR^j_{ld}\}_{j \in resolvedBy(D_i)} +$

	KDE	Mozilla
top 1 recall	30% - 50%	10% - 20%

Table 2. Performance of developers identification

$\{R^k_{notes}\}_{k \in commitedBy(D_i)}$, where CR^j_{sd} and CR^j_{ld} are respectively the short and the long description of the CRs the developer have fixed and R^k_{notes} is the notes the developer have included in his/her CVS commit operations.

Table 2 reports a summary of results we obtained in [4]. It shows, in a period of time, the percentage of CRs that in average has been assigned to developers retrieved by an information retrieval algorithm. KDE follows more than Mozilla the procedure suggested by the retrieval algorithm.

4. Concluding remarks and open issues

Software and change repositories give new opportunities to analyze how software changes and evolves. Textual information of software repositories is used to index relevant entities, source files and developers, of the software development process. As an initial application, we have made experiments aimed at predicting impacted source files and selecting the best candidate set of developers able to resolve a new change request.

An index of source files can also be used for categorizing source files about a topic that can, for example, justify or not the current subdivision of the system in modules and packages.

An index of developers can be used for developer competence classification. Trails left by developers in software repositories can be a valuable source of information to reconstruct their competence profiles: telling what you have done gives the opportunity to know who you are.

Software repositories provide new software perspectives that should integrate the classical source code perspective and can give to project managers and developers an effective support on maintenance and development activities. However many open issue should be faced, such as those regarding the quality of free text and project maturity.

Sometime CVS notes are used for communication rather that for description purpose and in almost all projects there is an initial period of transition that generates noise in both CVS and Bugzilla repositories.

Indexes can be build only for mature projects for which a huge amount of historical data is available. For young and immature projects both impacted files prediction and developers identification have failed. Using data of other similar mature projects can be an interesting direction of improvement. Other repositories rich of free text also used by the open source community are: mailing lists, newsgroups, and IRC conversations. In our experience these type of repositories are quite complex to analyze and require the development of ad-hoc approaches.

References

[1] Bugzilla. bug tracking system. http://www.bugzilla.org/.

[2] Cvs. concurrent versions system. http://www.cvshome.org/.

[3] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *METRICS '05: To appear in Proceedings of the 11th IEEE International Software Metrics Symposium*. IEEE Press, 2005.

[4] G. Canfora and L. Cerulo. Supporting change request assignment in open source development, 2005. Unpublished RCOST internal report, Available from the authors.

[5] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, 2001.

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance ICSM03*, Amsterdam, Netherlands, Sept. 2003.

[7] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[8] L. Lpez, J. Gonzlez-Barahona, and G. Robles. Applying social network analysis to the information in cvs respositories. In *IEEE 26th International Conference on Software Engineering - The International Workshop on Mining Software Repositories*, 2004.

[9] A. Michail. Data mining library reuse patterns using generalized association rules. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 167–176. ACM Press, 2000.

[10] B. Ribeiro-neto and Baeza-yates. *Modern Information Retrieval*. Addison Wesley, 1999.

[11] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining revision history. *IEEE Transactions on Software Engineering*, 30:574–586, Sept. 2004.

Workshop 2.2:
Workshop on Empirical Studies in Reverse Engineering

Session: Experimental Design

Characterization of Reverse Engineering Experiment Families

Marco Torchiano
Dipartimento di Automatica e Informatica
Politecnico di Torino
Italy
marco.torchiano@polito.it

Abstract

Within the large software engineering community there have been several attempts to provide a standard template to characterize empirical studies. The objectives being twofold: first improve the communication among researchers and second facilitate the discovery of potentially useful information for the practitioners.
In the emerging area of empirical studies in reverse engineering we could aim at a specialized template.

1. Introduction

The empirical branches of both “hard” and “soft” sciences have since a long time defined widely accepted standards for presenting and characterizing the result of their studies.
The advantages of such guidelines are:

- it is easier to understand a research paper if you know where to look for the information you are looking for;
- the search for the relevant studies is much easier since paper can be classified more easily;
- the general “validity” or “goodness” of a paper can be assessed with a quick browse;
- it is easy to identify blind spots of the studies and to design families of experiment to cover them.

In the software engineering community recently have been published some preliminary guidelines [5]. Also in a recent EU research project (ESERNET [3]) guidelines have been proposed on how to present and summarize empirical studies.
The problem of characterizing families of empirical studies in software engineering have been addressed in [2].

My intention in this position paper is to reason on how to adapt the proposals presented in the context of the wider empirical software engineering area to the narrower field of empirical studies in reverse engineering.

2. Characterization of RE empirical studies

In [2] we find a proposed approach to characterize families of experiment. In short the idea is to use the GQM [1] template to identify experiments.
There are five parameters in a GQM goal template:

1. Object of study: a process, product or any other experience model.
2. Purpose: to characterize (what is it?), evaluate (is it good?), predict (can I estimate something in the future?), control (can I manipulate events?), improve (can I improve events?).
3. Focus: model aimed at viewing the aspect of the object of study that is of interest, e.g., reliability of the product, defect detection/prevention capability of the process, accuracy of the cost model.
4. Point of view: e.g., the perspective of the person needing the information, e.g., in theory testing the point of view is usually the researcher trying to gain some knowledge.
5. Context: models aimed at describing environment in which the measurement is taken.

The goal is to adapt the above generic approach to characterize empirical studies in reverse engineering.
The objectives we wish to achieve are:

- make it easy to distinguish similar studies,
- identify clearly the essential features of each study,
- facilitate the identification of possible extensions in order to build experiment families

Examining the GQM template items we can find that some are more important than other, and we need some more information.

The *object* of the study is the fundamental classification feature; it represent the specific RE technique of method under study.
Perhaps with some restricted mindset we focus only on comparative studies. Therefore we can assume that the purpose is always evaluation.
The *focus* of the study let us understand which aspect of the RE technique we are interested in.
The point of view is usually that of the maintainer.
The *context* of the study must include the cofactors. This is very important to identify the external validity of the study and to enable the design of further experiments that could complement the original one.
It is also interesting to consider the *population* of the study. Since RE often employs automatic techniques, in those cases it can make more sense working with a population of programs than a population of maintainers (programmers, or developers).
Finally the main independent variable (or *factor*) of the study should be mentioned explicitly together with its values.

2.1. Example

Let us consider a couple of examples.

The first study is an experiment presented in [7] whose goal is to compare code annotations to drawing editor for generating documentation of object-oriented programs. The second study is a (planned) experiment to compare the completeness and accuracy of static versus dynamic analysis.
The characterization of the two empirical studies is presented in Table 1.

Table 1: Characterization of two sample studies.

	Study 1	**Study 2**
Object	Class diagram extraction method	Class model automatic extraction techniques
Focus	Ease of use	Model accuracy and completeness
Factor	METHOD = {annotations, drawing editor}	ANALYSIS TYPE = {static, dynamic}
Subjects	Maintainer	Programs
Context	Program characteristics: size, application domain, etc. Maintainer characteristics: experience, skill, etc. Tool	Program characteristics: size, application domain, etc. Tool

The context for both studies includes “tool”. For instance, in the first study we picked a drawing editor tool for convenience reasons. We can speculate that selecting another one the result would not change significantly but we cannot be sure.
In the first study we selected students as subjects. There is a never-ending discussion about the difference between students and professionals [6]. The only meaningful way of addressing this concern is to replicate the experiment with professional developers. Similar considerations hold for the program characteristics, for instance the size of the program can have a significant impact on the results [4].

3. Conclusions

The proposed template is derived from a sound empirical study characterization template. It addresses issues of clear identification and definition of extension points for empirical studies.
Since it has been derived though speculation on the basis of past experience with empirical studies, it would benefit both from an assessment and a discussion.

4. References

[1] V. Basili, G. Caldiera, and D. Rombach, "Goal question metric paradigm" in *Encyclopedia of Software Engineering*, vol. 1, J. J. Marciniak, Ed.: John Wiley & Sons, 1994.

[2] V. R. Basili, F. Shull, and F. Lanubile, "Building Knowledge through Families of Experiments" *IEEE Transactions on Software Engineering*, 25 (4): 456-473, 1999.

[3] R. Conradi and A. I. Wang Eds., "Empirical Methods and Studies in Software Engineering, Experiences from ESERNET": Springer, 2003.

[4] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics" *IEEE Transactions on Software Engineering*, 27 (7): 630 - 650, July 2001.

[5] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary Guidelines for Empirical Research in Software Engineering" *IEEE Transactions on Software Engineering*, 28 (8): 721-734, 2002.

[6] P. Runeson, "Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Student Data" Proc. of 7th International Conference on Empirical Assessment & Evaluation in Software Engineering (EASE'03), April 8-10, 2003

[7] M. Torchiano, F. Ricca, and P. Tonella, "A comparative study on the re-documentation of existing software: Code annotations vs. drawing editors" Proc. of IEEE International Symposium on Empirical Software Engineering (ISESE), Noosa Heads, Australia, November 17-18, 2005, pp.??

On empirical studies to analyze the usefulness and usability of reverse engineering tools

Tarja Systä and Kaisa Väänänen-Vainio-Mattila
Tampere University of Technology
Institute of Software Systems
{tarja.systa, kaisa.vaananen-vainio-mattila}@tut.fi

Various tools and techniques have been developed to support reverse engineering and program comprehension of software systems. In perhaps surprisingly many cases the tools have either been originally developed for, or their development has been greatly influenced by, a certain application example or software engineering team. This is, however, understandable since the reverse engineering tasks are often problem, language, and/or project-specific. This necessarily results in a set of greatly varying reverse engineering tools available, both in terms of their functionalities and visualization techniques used. That, in turn, makes is very difficult to compare the usefulness and usability of the tools. Without such comparisons and tool evaluations choosing the most appropriate tool can be quite challenging.

Due to the goal-driven nature of reverse engineering and program comprehension activities, the analysis steps differ case by case. The tools usually provide a set of functions for the software engineer to compose different views to the software, depending on a comprehension task in question. While such features greatly help the user, the understanding still need to be done by the software engineer herself. Namely, the actual reverse engineering or program comprehension activities can be at most semi-automated. This, again, is a challenge from the point of view of tool comparison with respect their usefulness and usability.

The variability of reverse engineering activities is also a challenge for the tool developers. How to build a reverse engineering tool that would be generally usable, when not only the programming languages, but also operations to be applied to analyze the software differ? In some tools this problem has been solved by providing a customizable and extensible tool that can be tailored to different needs[Rigi]. Customizability and extensibility clearly effect the usability and usefulness of the tool. Yet, in tool comparisons presented in the literature[Storey96, Storey97, Bellay98] this, to our opinion an important aspect, has been seldom emphasized.

The usability of SW engineering tools has not been extensively researched, and the same holds to reverse engineering tools [Lethbridgeea97, Mullerea00]. In many cases, the comparisons have been carried out in a more or less ad hoc manner, without proper scientific test set-up. The general metrics of usability as defined by ISO 9241-11 [ISO9241] – effectiveness, efficiency and subjective satisfaction – can be adapted to reverse engineering field. Effectiveness refers to the amount of problems that can be solved by using the specific tool and efficiency implies the amount of effort needed to solve the given reverse engineering tasks. Subjective satisfaction of using the tools is related to the professional users' satisfaction of using the tools in their daily work over a long period of time. In addition, the organisational satisfaction with the tools becomes a broader context for tool satisfaction. These realisation of these attributes in a SW tool differ significantly from the usability metrics of consumer products, where the emphasis is often on learnability, intuitiveness, aesthetics and short-term satisfaction and delight by an individual user.

In [Mullerea00] Muller *et al.* it is suggested that many investigative techniques and empirical studies may be appropriate for studying the benefits of reverse engineering tools. They suggest, e.g., expert reviews, user studies, field observations, case studies, and surveys and further present examples of their application for that purpose. Research approaches can also be designed according to the limited reported experiences of SW engineering tool research [Coppitea03]. In general, usability testing methods initially developed for consumer products or non-professional usage should be applied critically [Dicks02, Johnea97]. In specific, test cases must be real industrial cases to get valid results of effectiveness of the tools, and the usage experience must go beyond the learning curve in order to get reliable results of the efficiency and users' and organisational satisfaction.

Reverse engineering tools are seldom widely adopted in industry. One reason for that might be that they indeed are built to solve specific problems. Sometimes their poor scalability is also pointed out as a reason,

esp. when tried out with large industrial software systems. Tight integration of reverse engineering and forward engineering tools is also problematic, e.g. since different modeling notations are used. The integration would, however, be desirable since the reverse engineering activities are typically followed by forward engineering efforts.

Challenges of empirical studies on the usability and usefulness of reverse engineering tools can be identified as indicated above. Thus broader research set-ups will require extensive work, even developing new, more generic reverse engineering tools. Also, quality of software development is highly dependent on individual SW developers' skills and experience, and thus comparative studies require very careful selection of test users of the tools in order to get reliable results. Furthermore, a single test must last at least weeks, or even months and cover a broad set of SW development tasks. Control groups set-up must also be designed to get comparative results not only between different reverse engineering tools but also between user groups using and not using the tools.

While some empirical studies on comparing reverse engineering tools have been presented in the literature[Storeyea96, Storeyea97, Bellayea98], they have been mostly carried out without real tool users, e.g. with university students and small test groups. They also address no realistic industrial case study material and are carried out within a relatively short period of time. Yet, even such small empirical studies have unveiled significant flaws in some (possibly prototype) tools, e.g. severe parsing problems[Simea00]. Empirical studies in more realistic settings are needed, however, e.g. to discover scaling problems.

To get reliable results, the reverse engineering tools should be studied with the professional end users, with regards to the planned metrics, and by careful selected research methods in realistic test settings. Research method triangulation should be emphasised; both qualitative and quantitative approaches should be utilised.

References

[Bellayea98] B. Bellay and H. Gall, An evaluation of reverse engineering tool capabilities, *Journal of Software Maintenance:Research and Practice*, 10:305–331, 1998.

[Coppitea03] D. Coppit and K.J. Sullivan, Sound methods and effective tools for engineering modeling and analysis. *Proc. of the 25th international conference on Software Engineering*, IEEE Computer Society, 2003, pp. 198-207.

[Dicks02] R. S. Dicks. Mis-Usability: On the Uses and Misuses of Usability Testing, *Proc. of the 20th annual international conference on Computer Documentation,* ACM Press, 2002, pp. 26-30.

[ISO9241] ISO 9241: Ergonomic requirements for office work with visual display terminals. Part 11: Guidance on Usability, International Organization for Standardization, 1998.

[Johnea97] B.E. John and S.J. Marks, Tracking the Effectiveness of Usability Evaluation Methods, *Behaviour & Information Technology* 16 (4/5), 1997, pp. 188-202.

[Lethbridgeea97] T. Lethbridge and J. Singer, Understanding software maintenance tools: Some empirical research. In *IEEEWorkshop on Empirical Studies of Software Maintenance(WESS-97),* 1997, pp 157–162.

[Mullerea00] H. Muller, J. Jahnke, D. Smith, M.-A. Storey, S. Tilley, and K. Wong, Reverse Engineering: A Roadmap, In *Proc. of the 22nd International Conference on Software Engineering (ICSE'2000). Future of Software EngineeringTrack*, ACM Press, 2000, pp. 47–60.

[Rigi] Rigi Group Home Page, http://www.rigi.csc.uvic.ca.

[Simea00] S. Sim and M.-A. Storey, A Structured Demonstration of Program Comprehension Tools, in *Proc. of Working Conference on Reverse Engineering (WCRE),* 2000, pp. 184-193.

[Storeyea96] M.-A. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. M¨uller, On designing an experiment to evaluate a reverse engineering tool, In *Proceedings of the 3rdWorking Conference on Reverse Engineering (WCRE-96),* 1996, pp. 31-40.

[Storeyea97] M.-A. Storey, K. Wong, and H. M¨uller, How do program understanding tools affect how programmers understand programs, In *Proceedings of the 4th WorkingConference on Reverse Engineering (WCRE-97),* 1997, pp. 12-21.

Towards an Ontology of Factors Influencing Reverse Engineering

Bart Du Bois
Lab On ReEngineering
University Of Antwerp
bart.dubois@ua.ac.be

Abstract

In the context of the workshop's discussion on a general framework for empirical studies, this paper stresses the need for an ontology of factors influencing the application of reverse engineering techniques. Extracts from existing ontologies are used to illustrate the diversity of such factors. Accordingly, a methodology is required to guide the construction of the proposed ontology. This position paper aims to stimulate a discussion concerning the construction and refinement of a reverse engineering ontology as a valid asset in the composition of the general framework.

1 Introduction

In a controlled experiment on the support of a reverse engineering technique for bottom-up program comprehension we recently conducted, we have found that the effect of the technique covariated with a property of the source code being comprehended [Du Bois et al., 2005]. This covariation was noted as we incorporated variation of that particular property as a factor in the experiment.

Traditionally, empirical studies try to focus on some factors while keeping all others constant. Therefore, it is essential to have a clear overview of all factors affecting the process, technique or tool under study. Moreover, in order to reflect upon the application of the technique in a different context, it is essential to have a clear understanding of the nature and size of the effect of these factors.

An *ontology* is a hierarchical structured knowledge about things composed by subcategorizing them according to their essential (or at least relevant and/or cognitive) qualities. Therefore, ontologies can be used to structure the knowledge about factors affecting the result of empirical studies.

In [Kitchenham et al., 1999], four benefits that arise from the composition of an ontology are presented:

1. allow researchers to provide a context within which specific questions about maintenance can be investigated
2. help to understand and resolve contradictory results observed in empirical studies
3. provide a standard framework to assist the reporting of empirical studies in a manner such that they can be classified, understood and replicated
4. provide a framework for categorizing empirical studies and organizing them into a body of knowledge

As these benefits are in line with the objectives of the workshop, we propose to revise and refine existing ontologies to reach an ontology specific to the context of reverse engineering. This paper provides a first step towards the construction of a reverse engineering ontology by discussing existing ontologies and indicating the activities in constructing an ontology.

2 Existing ontologies

In this section, a number of ontologies in the context of software maintenance are discussed. These ontologies identify those factors for which variations influence empirical studies. Our purpose is not to provide a complete overview of existing ontologies, but to illustrate the diversity of these factors.

2.1 Ontology of software maintenance

The ontology of [Kitchenham et al., 1999] presents factors that might affect the result of empirical studies in software maintenance productivity, quality or efficiency. The ontology addressed factors related to (i) the maintained product; (ii) the maintenance activities; (iii) the software maintenance process; and (iv) staff involved in the maintenance process.

- **Maintained product**: size, application domain, age, maturity, composition, quality

- **Maintenance activities**: quality, type and age of input and output artefacts, human, hardware and software resources, product history, modification size and criticality, kind, requirements involved, configuration and event management, change control, development technology, paradigm and procedure ({method, script, technique})

- **Maintenance organization processes**: maintenance events and performance targets of service level agreements, formality and quality of configuration management, factors influencing the maintained product, investigation reports and modification activities

- **Staff**: attitude, motivation, skill and roles of the organization staff. Number and type of users, goals and employers of customers

2.2 Ontology of program comprehension strategies

[Storey et al., 1997] provide a description of the cognitive issues which should be considered during the design of a software exploration tool. These were categorized in (a) maintainer characteristics; (b) program characteristics; and (c) task characteristics, and were considered to be influences on program comprehension strategies of each of these categories are:

- **Maintainer characteristics**: application domain knowledge, programming domain knowledge, maintainer expertise, creativity, familiarity with the program and the CASE tool.

- **Program characteristics**: application domain, programming domain, program size, complexity and quality, availability of documentation and a CASE tool

- **Task characteristics**: task type, size and complexity, time constraints and environmental factors

These ontologies can be used as information sources during the construction of a reverse engineering ontology.

3 Constructing an ontology

[Jones et al., 1998] provide a survey of methodologies for building ontologies. They observed that methodologies seem to be split between mainly stage-based models and evolving prototype models. Both approaches typically produced an ontology two stages: a first informal description and then its formal embodiment in an ontology language. Activities appearing in most methodologies are:

- **Specification** of the purpose of the ontology, a description of its usage and users, its scope and the degree of formality required.

- **Data Collection**. While there is no prescribed elicitation method, proposed methods to construct the informal description include expert interviews, text analysis and and protocol analysis, expert interviews and brainstorming.

- **Conceptualization** of domain terms. This leads to a preliminary ontology containing so called proto-concepts or initial descriptions of kinds, relations and properties.

- **Integration** with other ontologies.

- **Formalization** in an ontology language (e.g. Ontolingua)

- **Evaluation** of completeness, consistency and redundancy.

Part of the specification activity is similar to that of [Kitchenham et al., 1999]. Therefore we can describe some of elements of the specification activity. The purpose of the ontology is to identify contextual factors that influence the results of empirical studies of reverse engineering. The ontology would be used in the construction of a general framework of such empirical studies. Its users would be both empirical researchers and industry partners interested in the effect of certain reverse engineering techniques. However, the determination of the scope is still unclear, and we hope that the discussions during the workshop will contribute to the specification of an ontology of reverse engineering.

4 Using an ontology in the construction of a general framework for empirical studies

There is no reason to assume that a reverse engineering ontology would exhibit a more restricted diversity and number of factors. Therefore, a procedure is required to introduce an ordering according to the relevance of these factors.

The properties of a factor are its nature, size and likelihood of variation. Most likely, factors included in the ontology will exhibit different levels in practice. The likelihood that a certain level of the factor will be exhibited in a representative set of cases can be described by its distribution. This distribution can be used to assess the relevance of including the factor in empirical studies. In case the distribution exhibits a large peak and short tails, it might not be necessary to assess the nature and effect of the factor at all.

Therefore, we propose to enrich the ontology with a description of the distribution of the likelihood of variation in each of the factors.

5 Conclusion

Results of empirical studies tend to form small islands of knowledge which are hard to relate to other studies. Therefore, this paper suggests to initiate a discussion regarding the construction of a reverse engineering ontology. The purpose of this ontology would be to identify factors which affect the effect of empirical studies. If such an ontology is used during the design of empirical studies, the results will be easier to integrate. Therefore, the ontology will prove a valuable asset during the construction of a general framework for empirical studies in reverse engineering.

References

[Du Bois et al., 2005] Du Bois, B., Demeyer, S., and Verelst, J. (2005). Does the refactor to understand reverse engineering pattern improve program comprehension? In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 334–343. IEEE.

[Jones et al., 1998] Jones, D., Bench-Capon, T., and Visser, P. (1998). Methodologies for ontology development. In *Proceedings IT and KNOWS Conference of the 15th IFIP World Computer Congress*, pages 62–75.

[Kitchenham et al., 1999] Kitchenham, B. A., Travassos, G. H., von Mayrhauser, A., Niessink, F., Schneidewind, N. F., Singer, J., Takada, S., Vehvilainen, R., and Yang, H. (1999). Towards an ontology of software maintenance. *Journal of Software Maintenance*, 11(6):365–389.

[Storey et al., 1997] Storey, M.-A. D., Fracchia, F. D., and Mueller, H. A. (1997). Cognitive design elements to support the construction of a mental model during software visualization. In *WPC '97: Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, page 17, Washington, DC, USA. IEEE Computer Society.

Workshop 2.3:
Workshop on Architecture Recovery towards Reuse

Towards System and Business Service Components through Reconnaissance and Reflexion

Jim Buckley, Andrew Le Gear
Computer Science and Information Systems Department,
The University of Limerick
Limerick, Ireland
e-Mail:{ jim.buckley/Andrew.Legear}@ul.ie

Abstract

Cheesman and Daniels have proposed an N-tier architecture for Component based systems comprising of, at least, a UI layer, a User-dialog layer, a System services layer and a Business services layer. This paper proposes reengineering legacy systems towards this architecture using a combination of 2 well known reengineering techniques: Reconnaissance and Reflexion.

Reconnaissance is initially used to identify the code involved in implementing the functional requirements of the system. Using the information generated, the software engineer identifies code-sets for potential System and Business service layer components. Finally, the software engineer uses Reconnaissance to refine the components suggested by the Reconnaissance analysis and to identify their interfaces.

1. Introduction

As software systems become increasingly large and complex [Corbi '89], the potential for developing systems from scratch has decreased [Meyer and Mingins '99]. Instead, software developers look for opportunities to re-use existing functionality (in the form of libraries, APIs or elements of existing systems). Accordingly, recovery towards re-use has become a central research theme in the area of Software Engineering [Erlikh 2000].

For current green-field projects, the situation with respect to re-use has been somewhat alleviated by the emergence of component-based development (CBD) [Szyperski] and the availability of deployment platforms, that support CBD. In addition, the explicit segregation of process-oriented and data oriented layers in the software architecture [McGurren 2004], [Galvin 2005] of component-based systems has further increased the potential for reuse. According to [Cheesman and Daniels 2001], the process-oriented layer (which they refer to as the System Services layer) is composed of components that model the system's use-cases, and the data oriented layer (which they refer to as the Business Service layer) is composed of components that model the data entities of interest to the system. Figure 1 illustrates this relationship.

However, in contrast to this green-fields scenario, mature software systems provide a dilemma for software organisations. Often they are the 'Golden Cow' of these organisations, providing a rich revenue stream and encompassing a wealth of business logic [Cody 2003]. However, the evolution of these systems over time can mean that their architecture and business logic becomes delocalized and obfuscated in an enormous code base [Eisenbarth et al. 2003], leading to difficulties in maintaining and reusing these core business assets.

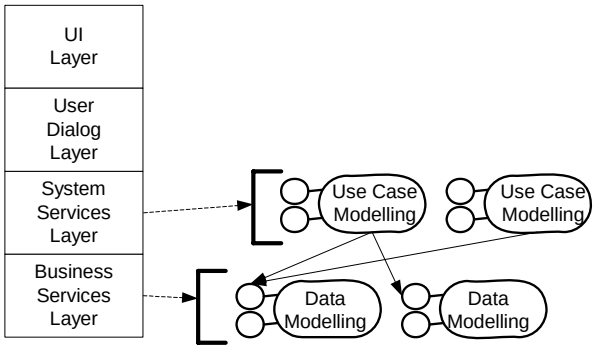


Fig 1: The N-Tier architecture proposed by [Cheesman and Daniels 2001]

In this paper we propose an integrated framework towards the recovery of System service and Business service layer components. Section 2 will discuss software reengineering and the two reengineering approaches that will be used in the proposed framework. Section 3 describes the process in detail. Section 4 places this framework in the context of the relevant literature, and section 5 concludes the paper.

2. Reengineering Approaches

Software reengineering has been defined as "the examination and alteration of a subject system to reconstitute it in a new form, and the subsequent implementation of the new form" [Chikovsky and Cross '90]. Unfortunately, reengineering legacy systems is far from trivial, and many organisations are reluctant to tamper

with existing code-bases to leverage their embedded value based on automated reengineering approaches alone. Instead, they prefer to use reengineering techniques to re-document their systems for software engineers who, in turn, can then use these views to re-structure the system.

In line with this observation, Koschke, in an evaluation of twenty-three software-clustering techniques, stipulates that domain expertise should be an integral part of component identification [Koschke 1999, Koschke 2000]. His papers also propose that reengineering should rely on several views of the software created from several different reengineering analyses.

Accordingly, in the process described here, documentation will be produced by static and dynamic reengineering analyses and the software engineer will use that data to propose System and Business service layer components. The role of the reengineering tools in this context is to *help* the engineer identify System and Business service components that are cohesive and exhibit a low level of dependency.

Over the years a multitude of reengineering approaches have been proposed to recover system architecture [Girard and Koschke '97] [Eisenbarth et al 2003] [Johnson 2002] and the business logic behind the implementation [Biggerstaff et al. '93], [Rich and Waters '88]. The architectural recovery techniques typically parse software systems statically for structural information. Guided by the software engineer and the relationships between the parsed structures of the code-base, these techniques aim to identify cohesive super-structures of low coupling in the system. Reflexion modelling (detailed in section 2.2) is one such approach.

Reengineering, for the recovery of business logic, is less well established. Reengineering approaches in this area have typically floundered on the complexity of the problem, either heading towards NP completeness [Rich and Waters '88], or circumventing the problem by limiting their scope to very narrow vertical domains [Quilici '96]. However, a dynamic analysis technique called Reconnaissance [Wilde and Scully '95] would seem to hold potential in this area. It has already been successfully trailed in an industrial context [LeGear et al. 2005a] for a related purpose. Both Reconnaissance and Reflexion are now described in greater detail.

2.1 Reconnaissance

Reconnaissance is a dynamic analysis re-engineering technique that was first proposed by [Wilde and Scully '95]. In this approach, various test cases for specific, observable functional requirements (Wilde refers to these as features) are executed using an instrumented version of the software system. The resultant set of traces or profiles show the source code that is executed for each functional requirement (See Figure 2). More formally, given a set of test cases, $T = \{t_1, t_2, \ldots, t_n\}$, a set of software elements, $E = \{e_1, e_2, \ldots, e_n\}$ and a set of features the system exhibits, $F = \{f_1, f_2, \ldots, f_n\}$ we can define two relations:

• EXERCISES : $T \times E \rightarrow$ BOOL, such that EXERCISES(t, e)= true if test case, t, exercises the software element, e, $t \in T$, $e \in E$,

• EXHIBITS : $T \times F \rightarrow$ BOOL, such that EXHIBITS$(t, f) = true$ if test case, t, exhibits the feature f , $t \in T$, $f \in F$,.

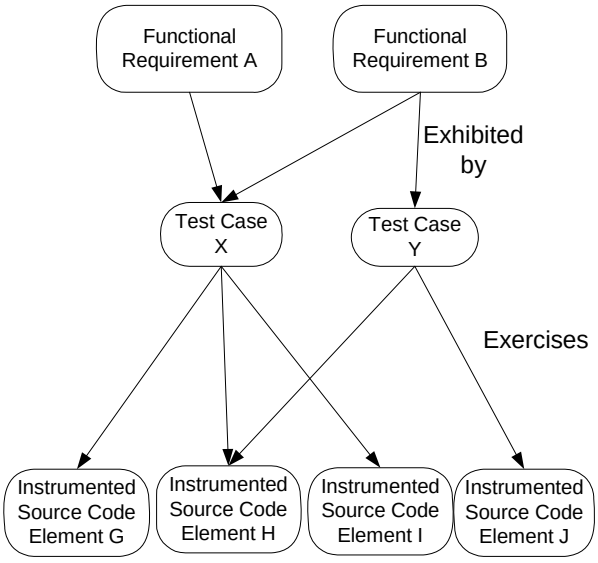


Fig 2: Relating Executing Code to Functional Requirements to Executed Code

Using the relations defined, several sets of source code elements may be calculated from retrieved coverage profiles [Wilde and Scully 1995]:

Involved Software Elements The set of involved software elements for a feature are those which are exercised in *any* test case exhibiting that feature. Given a feature, f, the set of involved software elements, IELEMS(f) may be calculated as:

$\{e{:}E \mid \text{for all } t \in T, (\text{EXHIBITS}(t, f) \wedge \text{EXERCISES}(t, e)) = \text{T}\}$

Indispensably Involved Software Elements The set of indispensably involved software elements for a feature are those which are exercised by *every* test case exhibiting that feature. Given a feature, f, the set of indispensably involved software elements, IIELEMS(f) may be calculated as:

$\{e{:}E \mid \text{for all } t \in T, \text{EXHIBITS}(t, f) \Rightarrow \text{EXERCISES}(t, e)\}$

Common Software Elements The set of common software elements are those elements in a system that are executed

by every test case profile gathered from the system. CELEMS generally represents utility code within the systems that is executed every time it is run [Wilde and Scully 1995]. Given a feature, *f*, the set of common software elements, CELEMS is calculated as:

$\{e : E \mid \text{for all } t \in T, (\text{EXERCISES}(t, e)=\text{T})\}$

Uniquely Involved Software Elements The set of uniquely involved software elements for a feature are those which are exercised by any test case exhibiting that feature but excluding any elements that are exercised in test cases that do not exhibit that feature. This is the same as the set of *involved software elements* except we exclude any software elements in that set that are exercised by test cases that do not exhibit that feature. UELEMS(f) has been shown experimentally to provide a useful starting point to begin searching when attempting to understand a particular functionality exhibited by a system [Wilde and Scully 1995]. Given a feature, *f*, the set of unique software elements, UELEMS(f), may be defined as:

$\text{IELEMS}(f) - \{e : E \mid \text{there exists } t \in T, (\neg\text{EXHIBITS}(t, f) \wedge \text{EXERCISES}(t, e))=\text{T}\}$

Shared Software Elements Using the sets defined in section 3.1 we can define the set of software elements shared by a feature, *f*, with other features as:

$\text{SHARED}(f) = \text{IIELEMS}(f) - \text{UELEMS}(f) - \text{CELEMS}$

This equation yields a set that is neither utility code nor unique to a feature, but software elements indispensably involved in a feature but also shared between two or more distinct features of a system. From a reuse perspective, the SHARED set gives a view of the software elements being reused by the running system [LeGear et al 2005a].

2.2 Reflexion

Reflexion modeling is a semi-automated, diagram-based, structural-summarization technique that domain experts can use to aid their comprehension of software. Introduced by [Murphy et al. 2001], the technique is primarily aimed at aiding software understanding and architectural recovery. The Reflexion-based architecture recovery process, illustrated in figure 3, executes as follows [Murphy and Notkin '97]:

1. The user, who is a domain expert, defines a *high-level model:* a conceptual model of the system, by reviewing available documentation, their knowledge and the source code.
2. An extraction tool is applied to the source code, to identify the elements of the system and their relationships. This is used to form the source model.
3. The user then defines a map, matching elements of the system with entities in their high-level conceptual model.
4. A Tool then computes the Reflexion model by comparing the source model with the user's high-level model, as described by the map. The output highlights the inconsistencies between the user's model and the source model.
5. The user uses the Reflexion model to target inconsistencies and refines his/her map or high-level model accordingly. He/She then re-computes the Reflexion model.
6. Steps 4 and 5 are repeated until the source and high-level models converge.

This software understanding method is different to previously proposed clustering techniques, due to its heavy user involvement, and is accompanied by promising results [Murphy et al. '95]. For example, one experiment on Microsoft Excel (1 million KLOC+) allowed an engineer to state that he gained a level of understanding of the code within one month that would normally have taken two years [Murphy and Notkin '97].

3. The Component Recovery Process

Considering [Cheesman and Daniels 2001]'s characterization of System and Business services layers, it is possible that the Reconnaissance sets will provide a preliminary insight into the architectural structure of the system. More specifically, Cheesman and Daniels consider the System service layer to be composed of components that model each use-case (equivalent to an EJB session bean in EJB parlance). These components then rely on Business service layer components that, model / provide wrappers to the data. These lower level components can be used by many System service level components and are typically less likely to change over time.

Mapping these observations back to the Reconnaissance sets, we can see that the Uniquely Involved software elements are associated with a use-case (observable functional requirement) and thus seem analogous to System service level components. In contrast Shared software elements are more likely to be associated with Business service components, as both get shared amongst different functionalities.

Admittedly, this is an in-exact approach. For example, several factors suggest that the mappings will be 'fuzzy':

- Cheesman and Daniels state in their methodology that a System service level component can be an aggregation of more than 1 use case. In this scenario, the individual sets of code, corresponding to the Uniquely Involved software elements, would have to be subsequently aggregated.
- It is possible that the same source code is used to implement two different use-cases and still not

belong to the Business layer. For example, a system service that reconciles the company's accounts with the bank, could be used as part of several payment processes (wages, invoice payment). In this case, elements of the shared set could belong to the System service layer.

- Shared software elements could also be associated with crosscutting concerns [] like error checking. In this scenario, the shared software elements would neither fall into the System nor the Business service layer.

So, this analysis, like other reengineering analyses should only be used as a cue towards the recovery of reusable components, and not as a definitive solution. Analyses that could be used to complement this approach include analyzing the calls to Database APIs, for the Business service layer, and analyzing callbacks to the GUI, for the System service layer.

The analysis proposed in this paper is Reflexion Modelling and in combination, these two techniques make up the proposed framework for the recovery of System and Business services layer components. Specifically, this is a four-staged process where:

- Reconnaissance is used to identify the indispensably involved, and uniquely involved source code for various system requirements. This data is then used to calculate the shared sets across functional requirements. Note that this analysis could be based upon the function testing performed by a quality control team. As part of their normal working activities, they could provide test scenarios, based on the functional requirements (use-cases) of the system, and these test cases could be used on an instrumented version of the system to produce the source code sets.
- The software engineer is then presented with some visualization of the three sets. In its simplest form, this visualization could be a listing of the source code elements in each set. The software engineer then uses these lists to form a first approximation of several System and Business service components. Business service components will be based on the Shared sets whereas System service components will be largely based on the Uniquely Involved source code (possibly augmented by the Indispensably Involved source code). This will, as discussed be an inexact approximation and other reengineering analyses should be used at this point to support this task.
- It is envisaged that Reflexion modeling will then be used to refine each provisional component recovered in this fashion. Here the software engineer generates a conceptual model of the system as two entities: the proposed component (mapping all the source code elements of the proposed component to its node) and the 'Rest-of-System' (mapping all the software elements not in the proposed component to the 'Rest-of-System' node). When the tool generates the Reflexion model, based on this mapping, it shows the coupling of the proposed component to the rest of the system. The software engineer can then refine the mapping to alter the set of source code elements included in or excluded from the component.

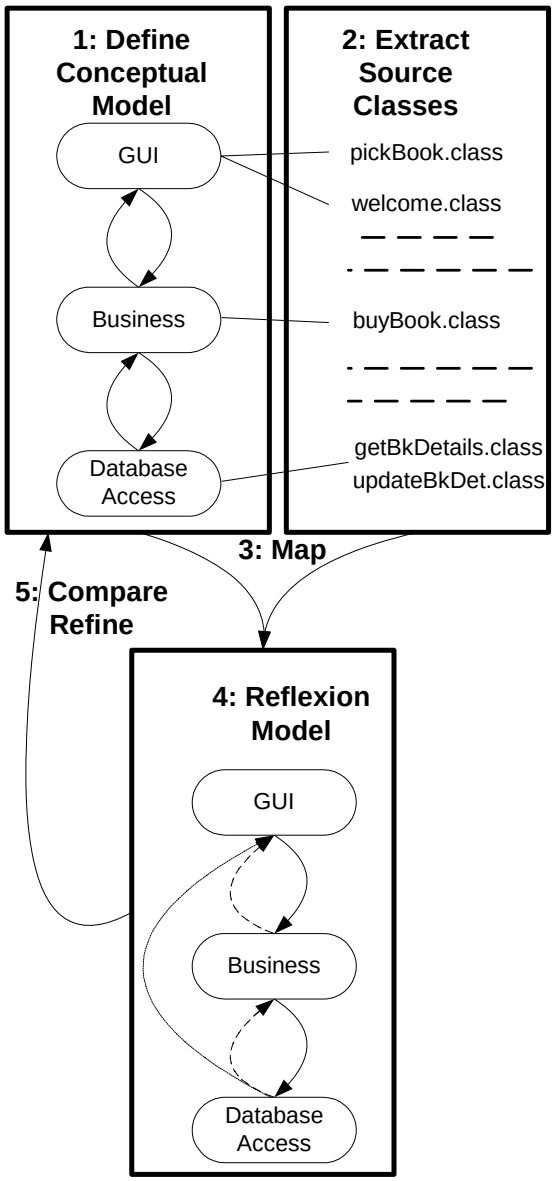


Fig 3: The Reflexion Modelling Process

- As the software engineer becomes satisfied with their component, they can choose to decompose the 'Rest-of-System' node, in order to identify the various interfaces of the component.

4. Related Work.

In other related work [Eisenbarth et al 2003] have expanded the dynamic analysis technique underpinning Reconnaissance to identify features, using concept analysis [Ball '99]. Concept analysis is a mathematical technique for describing binary relations. Central to concept analysis is the formation of a *concept lattice*. In terms of feature identification, this is a hierarchal, visual representation of the relationship between test cases and software elements [Eisenbarth et al 2001]. However, concept lattices cannot easily accommodate identification of a reuse perspective.

The work proposed here closely mirrors that of [Cimitile and Visaggio '95]. They use a graph-theoretical approach called dominance analysis to (in part) identify shared software elements in a call-graph representation of software systems. However, their reuse approach differs from ours in that they suggest grouping functions together based on static analysis. As such, there is no requirement-based perspective in their work and thus, they would be less able to identify System service layer components.

[Girard and Koschke '97] have advanced the work of Cimitile and Visaggio by taking data types and global data accesses into account. Instead of applying dominance analysis to a call graph representation of the system, they first aggregated nodes in the call graph that referred to the same atomic data-type or global variable in the system. Then they applied dominance analysis to the refined graph, in a manner analogous to Cimitile and Visaggio. We would like to apply their work to our research, using an analogous strategy where database accesses help:

- Cluster software elements into components;
- Discriminate between cross-cutting concerns and Business service components;

Aspect-oriented programming provides the ability to program and view software with specific concerns in mind [Kiczales et al '97]. Potentially then, the shared set could define aspects in the code, creating a viewpoint [IEEE 2000] on the system that may be used during aspect-oriented programming.

Part of the research proposed in this paper has been carried out already and has been reported on in several conferences. For example, the re-usability of the shared sets, in the context of the current systems evolution, has been reported on in [Le Gear et al 2005a] and component recovery from the shared sets has been reported on in [Le Gear et al 2005b]. Both these early industrial trials of the process were successful and suggest that the research avenue will be fruitful.

5. Conclusion.

This paper proposes a process towards System and Business service component recovery. In line with current state-of-the-art reengineering practice, it relies on both static and dynamic analysis, utilizes several analyses techniques and is interactively supported by the software engineer. Current work suggests that useful perspectives are achieved using this technique at the Business layer and future work will concentrate on expanding this work for the System layer.

Of course, if the approach can recover System and Business service components from static and dynamic reengineering analyses, then one is still left to ponder the composition of these components, so as to realise the business processes of the system. This is also an avenue for future work.

6. References

(Ball '99) T. Ball. The concept of dynamic analysis. In Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, pages 216–234, Toulouse, France, 1999.

(Biggerstaff '93). T.J. Biggerstaff. The Concept Assignment Problem in Program Understanding. In Proceedings of the 15th International Conference on Software Engineering. Pp 482-498.

(Cheesman and Daniels, 2001) Cheesman, J., and Daniels, J. UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley. 2001.

(Chikofsky and Cross '90) Chikofsky E.J. and Cross J.H. Reverse Engineering and Design Recovery. IEEE Software. Vol. 7, no. 1 pp 13-17

(Cimitile and Visaggio '95) Cimitile A. and Visaggio G. Software Salvaging and the Call Dominance Tree. Journal of Systems Software. Vol 28. no, 2. pp 117-127

(Cody 2003) Cody J.R. Comprehending Reality: Practical barriers to Industrial Adoption of Software Maintenance Automation. Proceedings of the 11th International Workshop on Program Comprehension. pp 196-206.

(Corbi ’89). Corbi T.A. Program Understanding: Challenge for the 1990’s. IBM System Journal. Vol 28. no 2. pp 294-306

(Eisenbarth et al. 2003) T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. IEEE Transactions on Software Engineering, 29(3):210–224, March 2003.

(Eisenbarth et al. 2001) T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In IEEE International Conference on Software Maintenance (ICSM’01), page 602, Universit¨at Stuttgart, Breitwiesenstr, 20-22, 70565, Stuttgart, Germany, November 7-9 2001.

(Erlikh, 2000). Erlikh, L. Leveraging Legacy System Dollars for E-Business. IEEE Computer, vol. 2, pp. 17-23, 2000.

(Galvin, 2005). Galvin, S. Enhancing the Role of Architectural Representations in Component-Based Development using Architectural Description Languages. Research MSc Thesis, Dept. of Computer Science and Information Systems, University of Limerick, June 2005.

(Girard and Koschke ’97) J-F Girard, R. Koschke. Finding Components in a Hierarchy of Modules: a Step towards Architectural Understanding. Proceedings of ICSM ’97. pp 58-65

(IEEE 2000) IEEE recommended practice for architectural description of software-intensive systems. Technical Report IEEE Std 1471-2000, 2000.

(Johnson, 2002). Johnson, P. D. Mining Existing Systems for Business Components: towards a Practical Method and Toolkit for the Evolution of Legacy Systems. In Procs. of the 26th Anuual International Computer Software and Applications Conference, Oxford, UK.

(Kiczales et al ’97) G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopez, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In Proceedings of European Conference on Object-Oriented Programming (ECOOP), June 1997.

(Koschke, 1999). Koschke, R. (1999). An Incremental Semi-Automatic Method for Component Recovery. In Procs. of the Conf. on Reverse Engineering, WCRE’99, Oct. 5-7 1999, Atlanta, Georgia, USA.

(Koschke, 2000). Koschke, R. Atomic Architectural Component Recovery for Program Understanding and Evolution, Ph.D. Thesis, Institute for Computer Science, University of Stuttgart, 2000.

(LeGear et al 2005a) A. LeGear, J. Buckley, B. Cleary, J.J. Collins K O’Dea. Achieving a Reuse Perspective within a Component Recovery Process: An Industrial Scale Case Study. Proceedings of the 13th International Workshop on Program Comprehension. pp 279-289.

(LeGear et al. 2005b) A. LeGear, J. Buckley, J.J. Collins, K. O’Dea. Software Reconn-exion: Understanding Software Using a Variation on Software Reconnaissance and Reflexion Modelling. International Symposium on Empirical Software Engineering. To appear.

(McGurren, 2004). McGurren, F. Supporting Component-Based Software Evolution through ConnX. Research MSc Thesis, Dept. of Computer Science and Information Systems, University of Limerick, June 2004.

(Meyer and Mingins ’99) Meyer B. and Mingins C.. Component-Based Development: From Buzz to Spark. IEEE Software. Vol 32. no 7. pp 35-37.

(Murphy et al. 2001) G. Murphy, D. Notkin, K.J. Sullivan Software Reflexion Models: Bridging the Gap between Design and Implementation. IEEE Transactions on Software Engineering. Vol 27, no. 4. pp 364-380

(Murphy and Notkin ‘97) G. C. Murphy and D. Notkin. Reengineering with Reflexion models: A case study. IEEE Computer, 17(2):29–36, 1997.

(Murphy et al. ’95) G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion models: Bridging the gap between source and high-level models. In Symposium on the Foundations of Software Engineering, pages 18–28, Washington D.C., 1995. ACMSIGSOFT.

(Quilici 93) A. Quilici. A Hybrid Approach to Recognizing Programming Plans. Proceedings of the 2nd International Workshop on Program Comprehension. pp 96-106.

(Rich and Waters ’88) C. Rich and R. C. Waters. The Programmer's Apprentice: A research overview. IEEE Computer, 21(11):10-25.

(Wilde and Scully ’95) N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. Journal of Software Maintenance: Research and Practice, 7(1):49–62, 1995.

Facilitating Architectural Recovery, Description & Reuse through Cognitive Mapping

Brendan Cleary and Chris Exton
Department of Computer Science and Information Systems
University of Limerick
Ireland
Brendan.Cleary@ul.ie, Chris.Exton@ul.ie

Abstract

Cognitive mapping is a quantitative text analysis technique which can extract from texts the portion of the author's mental model captured in that text at the time it was written. By applying cognitive mapping to texts produced by software engineers we can extract the segments of the mental models of engineers related to the systems for which they maintain responsibility. These mental models can then serve as the basis for establishing mappings between problem domain concepts captured in the cognitive map and solution domain concepts expressed in the code, facilitating architectural recovery, description and reuse.

1. Introduction

Ric Holt in his 2001 paper "*Software Architecture as a Shared Mental Model*" [1] builds on a trend towards stakeholder and concern centric architecture description [2] [3] [4] to describe software architecture as a shared mental model that system stakeholders use to reason and communicate about the systems for which they maintain responsibility. Assuming that such mental models do exist, how and what a system's stakeholders understand to be a systems architecture, that is their mental model, would impinge on the recovery, description and ultimately reuse of that architecture.

While how engineers understand or comprehend the systems they work with is frequently described in terms of processes, models or strategies [5] [6], it can also be described in terms of a persons ability to communicate intelligently in human oriented terms about a systems implementation. Biggerstaff describes a person as understanding a program when able to explain the program, its structure, its behaviour, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program [7]. This categorisation of how engineers understand software systems rests on different descriptions of "computational intent" [8] and the ability of the engineers to associate concepts appearing in one description of intent with the concepts in another.

In considering how engineers understand software systems we are usually concerned with two descriptions of intent, one described using a human language another using a programming language. For clarity and to compare related work, we refer to the former as problem domain and the latter as solution domain descriptions of intent. These different descriptions or domains of intent are separated by constraints on the sets of concepts expressible using the language in which they are described which constitute a "conceptual gap" between domains [9].

Approaches which attempt to bridge this conceptual gap for the purposes of architectural or design recovery, description or reuse such as tool assisted, lexical, statistical or dynamic analyses tend to rely on the parsing of solution domain artefacts (source code) to identify elements of the code base which are then inferred to be related to the implementation of some set of concepts from the problem domain. In this paper we present a complimentary approach towards bridging the concept assignment gap based on a quantitative text analysis technique called cognitive mapping [10]. By basing its analysis on problem domain artefacts, cognitive mapping allows the architecture of a system to be described from the perspective and using the language of the system stakeholders themselves. When combined with established lexical code analysis techniques our approach facilitates architecture recovery and reuse by establishing and maintaining a link between the solution domain artefacts (the source code) and the problem domain concepts which the code implements.

The next section (section 2) describes related work on concept assignment, section 3 describes cognitive mapping while section 4 briefly describes our application of cognitive mapping to bridging the conceptual gap between the problem and solution domains. Finally in section 5 we conclude and present ongoing and future work.

2. Related Work

Tool assisted concept assignment approaches facilitate users in manually or semi-automatically constructing models of the problem domain which are used to establish expectations to be confirmed in the solution domain and to record correspondences established by the user between the problem and solution domains [11]. These models can be constructed using techniques akin to those from domain engineering [12] or derived manually from developer's program investigation activities [13].

Lexical analysis based approaches are the simplest, most flexible and often most effective approaches used for identifying and establishing associations between problem and solution domain concepts [14] [15]. These approaches consider the solution domain artefacts as a corpus of texts in which they attempt to identify occurrences of problem domain concepts. For example Anquetil and Lethbridge have used a lexical analysis based on segmenting names of source files to identify potential problem domain concepts which are then used to cluster the source files as a method for architecture recovery [16].

Statistical and information retrieval based approaches also consider the solution domain artefacts as a corpus of texts to which they apply statistical analyses to generate short-form descriptors of the original artefacts. These short-form descriptors or profiles attempt to characterise the original artefact using a reduced set of concepts so that it can be easily compared with other profiles or queries [17]. While many of these techniques focus on attempting to derive traceability links between documentation and code [18, 19] [20] others are focused explicitly on the concept assignment problem [21] and feature location [22].

While tool assisted, lexical and statistical based concept assignment approaches tend to rely on problem domain concepts being encoded in the solution domain artefacts, through naming conventions etc, to establish mappings between the problem and solution domains [23]. Dynamic software analysis techniques such as software reconnaissance [24] or formal concept analysis [25], focus on localising concepts that are expressible either through test cases or through navigation of control and data flow.

These solution domain centric approaches to solving the concept assignment problem are considered desirable due to the accuracy and reliability afforded by solution domain artefacts and the ready availability of solution domain artefact parsing techniques. Unfortunately while a systems implementation may imply the intent that led to its development, the intent is not expressed explicitly in that implementation [26].

An alternate approach towards recovering lost intent is to combine existing analyses of solution domain artefacts with analysis of problem domain artefacts. Such an approach would allow us firstly to identify and secondly to define the problem domain concepts which we wish to associate with concepts in the solution domain from the perspective and in the language of the engineers' expert in the system.

3. Cognitive Mapping

A mental model is the model people have of themselves, others, the environment, and the things with which they interact, formed through experience, training and instruction (Norman 1988). Based on the assumption that language and knowledge can be modelled as networks or maps of words and the relations between them [27], texts can be thought of as containing a portion of the author's mental model at the time the text was created [10]. Working under the assumption that the meaning of a text does not result from single words but from the co-occurrence of different words [28], cognitive mapping is a quantitative text analysis technique that systematically extracts and analyses the links between words in a text in order to model the authors mental or cognitive map as networks of words [29] [30] [31]. This map is then hypothesised to approximate a portion of the mental model of the texts author at the time the text was composed [32].

While current general purpose programming languages do not allow for the direct expression of programmer intent [12] [26], software engineers have long used other software artefacts such as requirements, architectural and design documentation and even email to express concerns which cannot be expressed directly in the source code. Analysing these texts using cognitive mapping allows us to extract and make explicit the portion of the software engineer's mental model relative the system under study expressed within as maps of concepts, thus capturing and making explicit some of the original intent of the engineer. These maps can then be bound, using established concept assignment approaches, to the code base or other solution domain artefacts to facilitate recovery, description and reuse.

In pursuit of these goals we have developed an opertaionalised cognitive mapping procedure (Figure 1) based on Carley and Palmquist's approach [10] consisting of a semi-automated concept set definition phase, an automated map construction and a automated concept assignment phase which we describe briefly next.

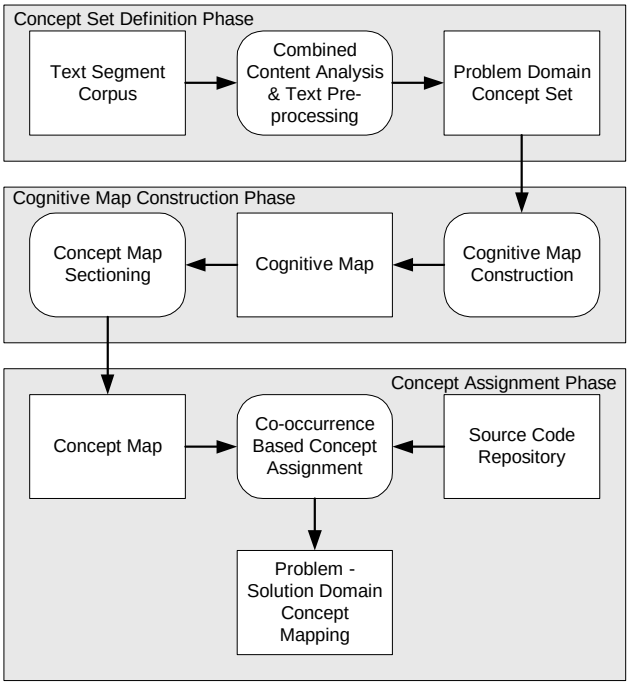


Figure 1 Concept Assignment through Cognitive Mapping Procedure

3.1 Concept Set Definition

After compiling a corpus of text segments from system documentation, emails or transcripts of engineer interviews, our concept set definition phase identifies a set of concepts from the corpus using a combined manual content analysis and automatic text pre-processing approach.

3.2 Cognitive Mapping

This set of problem domain concepts then forms the basis on which conceptual maps are constructed using a windowing based approach, which creates statements between concepts in text segments which co-occur within the window.

3.3 Concept Assignment

Finally the concept assignment phase sees the cognitive maps sectioned to create concept maps which are then used to establish correspondences between problem domain and solution domain concepts through simple lexical matching of concepts in the concept maps with concepts occurring in the code base, taking advantage of latent intent captured in the implementation such as naming conventions and comments. We briefly describe this final phase next.

4. Concept Assignment through Cognitive Mapping

Cognitive mapping generates for a system, a set of stakeholders and a set of text segments authored by those stakeholders a map of concepts that describe that system from the perspective of the stakeholders. A cognitive map can be defined simply as a set of concepts and a co-occurrence relation defined over that set.

The set of concepts C_p, contains all those problem domain concepts identified using the cognitive mapping procedure outlined above.

- $C_p = \{c_1, c_2, ..., c_n\}$

The relation $COOCCURS_p$ defines the co-occurrence relation between concepts in the set of concepts C_p.

- $COOCCURS_p : C_p \times C_p \rightarrow BOOL$, such that $COOCCURS_p(c_1, c_2)$, $c_1 \in C_p$, $c_2 \in C_p$ is true if c_1 co-occurs with c_2 within a given window w in the set of problem domain artefacts.

Cognitive maps can be sectioned to create concept maps, each of which describes a concept in terms of that concepts relationship to other concepts occurring in the cognitive map [31]. A set of focus concepts $F \subseteq C_p$ can be used to slice through the cognitive map and generate a concept map consisting of the set of concepts $S_p \subseteq (C_p - F)$ which co-occur with the set of focus concepts.

- $F = \{f_1, f_2, ..., f_n\}$
- $S_p = \{c \in C_p \mid \forall f \in F, COOCCURS_p(c, f)\}$

These concept maps can be used to localise problem domain concepts to a set of solution domain concepts, by matching the concept relationship patterns of a problem domain concept to corresponding patterns of concept associations in the source code. As concept maps provide a more expressive definition for a set of concepts under study F, defined in terms of patterns of relationships to other concepts S_p the degree of correspondence between the co-occurrence sets for a set of focus concepts in the problem domain artefacts and in the source code can be

used as a measure of confidence that a set of matched solution domain concepts are related to the implementation of a problem domain concept. This approach should then be considered as a refinement on existing lexical analysis concept assignment approaches that uses the context in which a concept occurs in the problem domain (S_p), to establish the veracity of lexical matches.

5. Conclusions & Future Work

We have proposed that the recovery, description and reuse of a systems architecture should take into account what the engineer's responsible for that system understand to be its architecture. Our research looks to a quantitative text analysis technique called cognitive mapping to create models that describe the concerns that software engineers maintain about their systems using the language and concepts which the engineers themselves use to describe those concerns. We then bind these models, through a process of concept assignment using lexical analysis, to the elements of the code base that implement those concerns. As such we aim to create views of a systems architecture which describe that system from the perspective and using the language of the engineers themselves. When combined with tool support these views, overlaid over the code base, enable engineers unfamiliar with the system or engaged in architectural recovery to capitalise on the expert knowledge or intent captured in the models in localising and understanding the implementation of problem domain concepts and concerns.

Currently we are conducting two case studies using the techniques described here, the first on an experimental visualisation framework [33], the second on a commercial ERP system. These case studies will allow us to refine and quantify the performance of the technique in a real world setting. We are also developing visualisation tool support that will see the mapping between concept maps and the source code presented to the user through the eclipse IDE [34].

6. References

[1] R. Holt, "Software Architecture as a Shared Mental Model.," presented at ASERC Workshop on Software Architecture, University of Alberta, 2001.

[2] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, Documenting Software Architectures, 1st edition ed: Addison-Wesley Professional, 2002.

[3] IEEE, "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems," in IEEE Std 1471-2000, S. E. S. Committee, Ed., 2000.

[4] M. Fowler, "Design - Who needs an architect?," Software, IEEE, vol. 20, pp. 11-13, 2003.

[5] N. Pennington, "Comprehension strategies in programming," presented at Empierical Studies of Programmers: Second Workshop, New Jersey, 1987.

[6] J. Good, "Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension," University of Edinburgh, 1999.

[7] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," Commun. ACM, vol. 37, pp. 72-82, 1994.

[8] C. Simonyi, "Intentional Programming," The Intentional Software Corporation, 2005.

[9] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," presented at Program Comprehension, 2002. Proceedings. 10th International Workshop on, 2002.

[10] K. Carley and M. Palmquist, "Extracting, Representing and Analyzing Mental Models," Social Forces, vol. 3, pp. 601-636, 1992.

[11] R. Clayton, S. Rugaber, and L. Wills, "Dowsing: a tool framework for domain-oriented browsing of software artifacts," presented at Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on, 1998.

[12] K. Czarnecki and U. Eisenecker, Generative Programming - Methods, Tools and Applications: Addison-Wesley, 2000.

[13] M. P. Robillard, "FEAT - Feature Exploeration and Analysis Tool," http://www.cs.ubc.ca/labs/spl/projects/feat/, 2004.

[14] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," presented at Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, Toronto, Ontario, Canada, 1997.

[15] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds, "A comparison of methods for locating features in legacy software," Journal of Systems and Software, vol. 65, pp. 105-114, 2003.

[16] N. Anquetil and T. C. Lethbridge, "Recovering software architecture from the names of source files," Journal of Software Maintenance, vol. 11, pp. 201-221, 1999.

[17] G. Salton, Automatic Text Processing The transormation analysis and reterival of information by computer, 1989.

[18] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," Software Engineering, IEEE Transactions on, vol. 28, pp. 970-983, 2002.

[19] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Information retrieval models for recovering traceability links between code and documentation," presented at Software Maintenance, 2000. Proceedings. International Conference on, 2000.

[20] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," Software Engineering, IEEE Transactions on, vol. 17, pp. 800-813, 1991.

[21] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," presented at Reverse Engineering, 2004. Proceedings. 11th Working Conference on, 2004.

[22] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a Static Non-Interactive Approach to Fea-

ture Location," presented at International Conference on Software Engineering, Edinburgh, Scotland, 2004.
[23] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in Proceedings of the 15th international conference on Software Engineering. Baltimore, Maryland, United States: IEEE Computer Society Press, 1993, pp. 482-498.
[24] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," Journal of Software Maintenance: Research and Practice, vol. 7, pp. 49-62, 1995.
[25] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," Software Engineering, IEEE Transactions on, vol. 29, pp. 210-224, 2003.
[26] J. Greenfield and K. Short, Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools: John Wiley & Sons, 2004.
[27] J. F. Sowa, Conceptual Structures: Information Processing in Mind and Machine: Addison-Wesley., 1984.
[28] Z. Cornelia and L. Juliane, "Computer-assisted Content Analysis without Dictionary," presented at Sixth International Conference on Logic and Methodology - Recent Developments and Applications in Social Research Methodology, Amsterdam, The Netherlands, 2004.
[29] E. T. Lewis, J. Diesner, and K. M. Carley, "Using Automated Text Analysis to Study Self-Presentation Strategies," presented at Computational Analysis of Social and Organizational Systems (CASOS) Conference, Pittsburgh PA, 2001.
[30] J. Diesner and K. Carley, "Using Network Text Analysis to Detect the Organizational Structure of Covert Networks," presented at NAACSOS, 2004.
[31] R. Popping, Computer-assisted Text Analysis. London: Thousand Oaks: Sage Publications, 2000.
[32] K. M. Carley, "Extracting team mental models through textual analysis.," Journal of Organizational Behavior, vol. 18, pp. 533-558, 1997.
[33] B. Cleary and C. Exton, "CHIVE - a program source visualisation framework," presented at 12th IEEE International Workshop on Program Comprehension, Bari, Italy, 2004.
[34] The eclipse project, "eclipse," www.eclipse.org, 2005.

Reusing Code for Modernization of Legacy Systems

Meena Jha
School of Computer Science and Engineering,
The University of New South Wales,
Sydney NSW 2052, Australia
e-Mail: meenaj@cse.unsw.edu.au

Piyush Maheshwari
The University of New South Wales and
National ICT Australia Ltd.,
Sydney NSW 2052, Australia
e-Mail: piyush@cse.unsw.edu.au

Abstract

Modernizing scientific legacy system to object-oriented platforms has been a challenge for research community over the past few years. The invaluable business logic of legacy system represents many years of coding, developments, real-life experiences, enhancements, modifications, debugging etc. It is infeasible to rewrite the entire software with new design rules. Reusing is perhaps the best strategy to handle complexities of software development. Architectural design pattern in conjunction with generic programming is a very powerful technique, which emphasises upon reusing software. We believe that design patterns could be applied to develop very flexible systems in which old design can be restructured and codes could be reused in some classes. Object-oriented programming enables us to organize the system into objects by information hiding, encapsulation, polymorphism and other techniques, which results in increased flexibility compared to non object-oriented systems. In this paper, we propose legacy code reuse for modernization of scientific legacy system. Moreover, the paper proposes how scientific computing community can change programming paradigm to take an advantage of modern software design principles, particularly object-oriented programming.

1. Introduction

Software reuse is identified as one of the best strategies to handle complexities associated with development and modernization of complex legacy software. Design pattern and generic programming can be employed to take the advantages of legacy systems. A number of legacy systems are mostly written in 3GL programming languages such as COBOL, RPG, PL1, FORTRAN, BASIC, PASCAL, C, etc. [1]. The changing technology is pushing the modernization of legacy system in several ways.

Many approaches to modernize legacy systems have been developed. The list of related work would be too long. However, the state of the art may be found in some recent publications [1, 2]. The current situation in legacy system modernization can be summarized as follows:

- Database reverse engineering is sufficiently mature to be applied in practice [8].
- Wrapping solution are short-term solutions that can complicate legacy system maintenance and management over time [1].
- Redevelopment approaches are considered risky for most organizations [1, 3].
- Most of the database reverse engineering literature examines solutions for migration of relational databases.
- There is a lack of literature on successful modernization processes. Many modernization project fail as outlined by the Standish Group [3].
- The reverse engineering of procedural components of a large application is still unsolved [15].

Modernization can be done at different levels. At lower levels, modernization takes the form of transforming the code from one language into another. At higher levels, the structure of the system may be changed as well as to make it, for instance, more object-oriented. At still higher levels, the global architecture of the system may be changed as part of the modernization process. Although, design pattern and generic programming have been very successful in new software development, we are not aware of any work from other research groups, who have studied the effectiveness of these techniques in restructuring the legacy codes and hence modernizing legacy systems. Most of the users when encounter the word *reusable software code*, they immediately start thinking of Black Box approach. The word reuse in software has intrinsic character of being tailored according to one's need. Reuse does not prohibit customization, but emphasizes on increasing productivity by learning from experiences of others and apply them judiciously. The reusable software code should increase the quality of the software. There are no universally accepted metrics, which can be used to measure the quality in an objective ways. Fenton (1991) described software quality as (Reliability + Availability + Maintainability + Usability) and Maintainability has been described as (Understandability + Modifiability + Extendability + Testability).

The paper is structured as follows. Section 2 describes our classification of current modernization approaches used in legacy systems. Section 3 describes our approach to legacy system modernization from procedural to object-oriented paradigm. Finally Section 4 summarizes the conclusion.

2. Classification of Current Modernization Approaches

The first distinction made is the 'platform change' decision, because that characterizes the most important decision one has to make when planning a legacy system modernization strategy. By 'platform change' we mean changing the old procedural programming language to a component based design. The 'no platform change' means staying with the current environments upgrade solution. If no platform change is decided then we are limited to the approaches that can all be categorized as 'wrapping'.

Wrapping Approach: Wrapping approach is divided into three alternative approaches [2].

- 'User Interface Wrapping', where screens of the legacy system are captured and are mapped into a modern graphical interface.
- 'Data Wrapping', where new interfaces are developed for legacy data structures. Therefore the old data can be accessed or made available to new applications. This approach itself could be subdivided into 'database gateways', 'XML integration' and 'data replication' methods.
- 'Function Wrapping', where not only the data but the business logic encoded in old legacy programming language are wrapped and accessed through an interface by other applications. It could be further subdivided into three subclasses. They are component wrapping, object-oriented wrapping and CGI integration. Object-oriented wrapping and component wrapping can also be used in conjunction with reverse engineering approaches [2].

Redevelopment Approach: Redevelopment approach can be categorized in two main classes. The first class is 'engineering from scratch' that is the most radical solution to legacy problems. This approach is based on leaving the existing code behind and rewriting the whole system from scratch. The second class is 'reverse engineering', which can be subdivided into two subclasses as: white-box and black-box approaches [1]. Black box approach is often based on wrapping, surrounding the legacy system with a software layer that hides the unwanted complexity of the legacy system and exports a modern interface. Unfortunately, this solution is often not practical, and often requires understanding the legacy system internals using white-box techniques [10]. In this subclass of reverse engineering, legacy system is kept intact and is being treated as one monolithic problem. This leads us to the 'function wrapping' techniques, but in the context of a new platform. The object oriented and the components wrapping approaches are more adequate to be used in conjunction with reverse engineering. Using these approaches, the interfacing logic is to be extracted through a shallow analysis of interconnected components, when possible. This objective will be to transfer the legacy components to the new platform and re-interface and restructure the links between them. However, translating the monolithic procedural legacy programs to a rich hierarchic and structured semantic of independent components can be a very difficult task, if not impossible [2, 3]. Figure 1 describes redevelopment approaches.

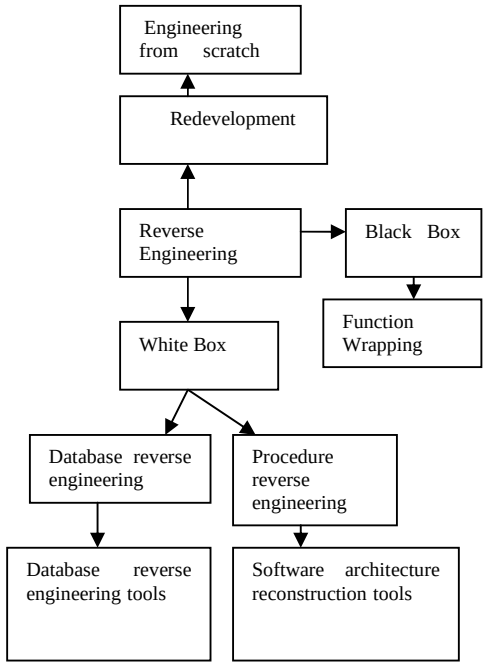


Figure 1: Classification of redevelopment approaches

In white-box subclass, the business-logic is to be extracted through deep analysis of legacy system. This subclass can be separated into two domains of 'database reverse engineering' and 'procedure reverse engineering'. The database reverse engineering can be handled using database reverse engineering tools [8]. For procedure reverse engineering, it requires analysing and understanding the architecture of existing systems to enable modification of the architecture to satisfy new requirements and eliminate software deficiencies [11].

Migration Approach: It can be divided into two classes. The first class is 'component migration' in which the large legacy systems are broken down into independent components and each component is migrated separately [3]. There will be a period of transition where both legacy and the new platform have to be online and work together. Two strategies will arise, 'phased interoperability' and 'parallel operations'. Both strategies need the data to be shared via database gateways, or replicated on the two platforms, or sliced into separate independent domains to be migrated gradually to a new platform [2, 3]. The data slicing is not easy to apply to legacy database.

The second class migration approach is the 'system migration' approach in which the whole legacy system and data are transferred to a new platform in a single step. There are two subclasses to this approach. The first one is the 'no value added migration' approach that can be done by 'platform emulation' or 'straightforward transformation' The platform emulation approach consists of moving the whole legacy system to 'virtual legacy machine' emulated on a new platform. Such solution does not bring any progress to the old system other than switching to a newer hardware when the old hardware is no longer supported. On the other hand, 'the straightforward transformation' of all the legacy components is not applicable to all legacy systems. The second subclass to the 'system migration' approach is the 'value-added modernization'. This approach leads to changes in the 'user interface', the 'database', and the 'program code'. Although, modernization may be more complex than wrapping in this situation, its long-term benefits will be greater.

From a detailed review of the literature in this area there has been no successful practical experience report from projects using a comprehensive modernization approach. The use of these approaches is still somewhat risky and advanced.

3. Our Approach to Legacy System Modernization

For many applications within the science and engineering community the root implementation language was and still is FORTRAN 77 and for some, even FORTRAN 66. Software engineering has developed and languages have grown and now FORTRAN 2003, C, and C++ provide the main modern vehicles for these applications. To maintain and continue to develop the science encapsulated in the code a process of transformation and re-engineering must be formalized and undertaken. This process can be broken into following four basic phases:

- Phase 1: **Analyse the legacy application**
- Phase 2: **Reconstruction of Legacy system**
- Phase 3: **Design Structure**
- Phase 4: **Transformation (Procedural ->OOP)**

Phase 1: Analyse the legacy application

This phase analyses legacy systems to capture their structure and to identify problems caused by the past development and evolution. The systems will have to be analysed and the results represented in a form suitable for further analysis. The analysis includes identifying the system components. This task includes gathering all application 'artifacts' such as source code, copybooks, JCL etc., statistics about the size, complexity, amount of dead code or unused code, and amount of bad programming for each program. We must clean up the "spaghetti": the replacement of computed GOTO statements with IF-THEN-ELSE blocks. Probably one of the most undesirable FORTRAN features is the COMMON block. As the only mechanism for providing global data it has been heavily use in most FORTRAN programs. FORTRAN 90/95 provides a versatile and safe mechanism for making global information available to program units through the MODULE. FORTRAN 90 can mimic some object-oriented features through combinations of its TYPE and MODULE syntax elements. The USE statements removes the need for the INCLUDE statement to make the global available consistently throughout the program. The construction of an equivalent MODULE block for a COMMON block is quite straightforward. However if the program contains a large number of COMMONs this could be a daunting task. Fortunately many software tool suites provide this conversion process.

Get the base code into a conforming form. As mention above often the legacy code are in FORTRAN 77 or 66 or even worse a mixture of standards and dialects. In general the standard of research programming is limited and most often leads to not particularly portable software. The developers often adopt mechanisms from other languages: the main example of this is # directives from the Unix C pre-processor cpp. #include, #define and #ifdef are some of the culprits. The use of pre-processor as a configuration tool allows the maintenance of multiple configurations in a single file. However as far as software engineering tools are concerned statements such as #define and #ifdef, that drive preprocessors such as cpp and fpp, are not part of the FORTRAN language and lead to syntax errors being flagged.

Once the basic code is in standard form automated transformation tools can be used to change the software format. This can be purely a source-to-source transformation and no structural changes are made. These types of transformation are often thought of as pretty printing. However some transformation tools will perform limited re-structuring of the program as it is being transformed. We will be using *Understand for FORTRAN* tool. It is an interactive tool providing reverse engineering, automatic documentation, metrics and cross-referencing of FORTRAN source code. *Understand for FORTRAN* analyses FORTRAN source code to create a repository of the relations and structures contained within it. The repository can help us answer questions such as:

- What is this entity?
- Where is it changed?
- Where is it referenced?
- Who depends on it?
- What does it depend on?

Deliverables: Set of legacy system analysis techniques and definition of relationships between system elements.

Phase 2: Reconstruct the architecture

This phase reconstructs legacy systems to a structure suitable for best practice economic evolution of the legacy system. This phase of the modernization would discover the design of the legacy system. Continued maintenance has shown that the original structure of the system design has usually been lost. The source code hosts the most current information about the system. If business rules are to be clustered into concept services then they need to be reconstructed to undo degradation process. The software architecture can be fully understood with the help of software architecture reconstruction (SAR). The importance of SAR has been established in understanding the legacy system. Architecture Reconstruction Mining (ARMIN) tool could be used to analyse the legacy system. ARMIN is a tool developed by the Software Engineering Institute and Robert Bosch Corporation. Use of ARMIN in architecture reconstruction has been reported in [5]. It is highly configurable modeling and visualization tool, recognizing inputs in RSF [14], based on an element and relationship schema The end result of the architectural reconstruction process is a set of architectural views.

The reconstruction operates on views to reveal broad, coarse-grained insights into the architecture. Reconstruction consists of two primary activities: *visualization and interaction* and *pattern definition and recognition. Visualization and interaction* provides a mechanism by which the user may interactively visualize, explore and manipulate views. SAR generates source model and architectural views. The visualization of architectural view can help finding out the answers of the question, which can form the basis of the restructuring. Some of the questions include:

- What are the subsystems or components of the software systems?
- How should the interfaces between components be structured?
- What are the characteristics of the component communication?

The architecture reconstruction effort can produce useful architectural metrics and views. Figure 2 shows the steps involved in SAR.

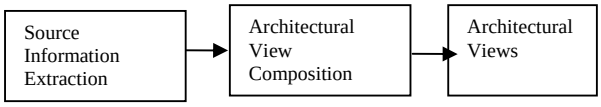


Figure 2: Reconstruct the architecture

Deliverables: A set of techniques for reconstructing legacy systems for restructuring.

Phase 3: Design Structure: Restructuring

This phase is speculative and will attempt to propose a structure for modernized systems for future evolution. Restructuring improves the quality of software. Since our focus is on changing the programming paradigm reusing legacy code from procedural to object-oriented, we use two commonly referenced structural attributes as restructuring criteria. First is cohesion and second is coupling. The restructuring should not change, add or delete functions.

Figure 3 describes the design structural phase. It consists of following steps:

1. Architectural views can be captured and extracted from source code. We can automate the extraction of information.
2. The restructuring step manipulates only design information rather than implementation details. We restructure design structures by decomposing and composing them.
3. The design structure is represented in a visual form. We can thus visualize the whole phase to help us understand the design structure.
4. An *architectural pattern* expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

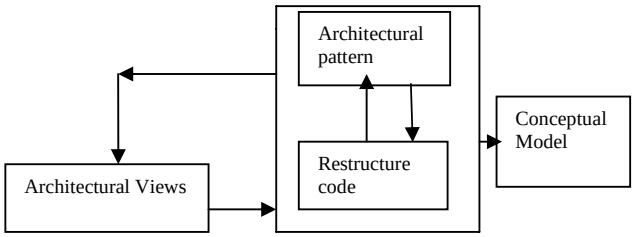


Figure 3: Design structure phase

The restructuring process consists of a series of semantic preserving decompositions and compositions of 'processing elements'. If the functions are in the same logical unit then through abstraction and grouping of the functions within the unit then ARMIN can be used to generate a view that shows the logical connection. If the functions are in different logical units but have relationships for example there are calls between them or they share data then through a different abstraction it would be possible to generate a view that shows the connection between the logical units and the calls that make up that connection and also abstraction through a data view could show a set of functions related to some logical grouping of data. Restructuring is completed by applying restructuring operations. We use the following eight basic operations [12]:

1. Coincidental Decomposition: A module exhibiting coincidental cohesion has disjoint components –one or more groups of data tokens linked to individual outputs without any dependence relations on another group. Separating the disjoint groups can easily spill these modules.
2. CIC Decomposition: Modules with conditional, iterative, or communicational cohesion have one or more inputs tied to all of the outputs. Copying all data

tokens linked to more than one output can decompose these modules.

3. Sequential Decomposition A: In modules with sequential cohesion one or more outputs depend on other outputs. Splitting the modules into two or more modules with sequential coupling, which is one of the most desired forms of coupling, can decompose such modules.
4. Sequential Composition: The inverse of sequential decomposition is to compose two modules with sequential coupling.
5. Sequential Decomposition: An alternative to Sequential Decomposition A is to replace an output component with a module call, and move the output components and the components that it depends on into a separate callee module.
6. Caller/Callee Composition: Two modules can be composed if they have a call relation, exhibit either conditional or computational coupling, and the callee has only one caller. The call statement is replaced by the tokens of the callee, and unnecessary coupling is reduced.
7. Hide: The hide operation, converts an exported output into a hidden local variable, when the exported output is not actually used externally.
8. Reveal: Reveal is the inverse of hide. Using reveal, a local variable is exported by changing the local variable into an output variable; Reveal can be used to separate a hidden function from a large module.

Deliverables: Refined OO model. Collection of related business rules and conjecture on how systems may evolve and what structure they should have.

Phase 4: Transformation

Once the stage of reconstruction and restructure is complete we can move on to look at the use of object-oriented techniques, which can, in the long term, improve the maintenance and future development of the program. The use of object-orientation and languages such as C++ is becoming more common in numerical software. It would be clearly an advantage for many if there were tools to perform the direct translations of one programming language into another: FORTRAN 77 to C++ for example. Clearly this is currently an impossible task. Translating the statements of a simple procedural language into one that is rich in object orientation, user definable operators and user definable data types is a tall order. In transformation phase object oriented programming features would be handled. We would be working on to identify object-oriented features from procedural software.

FORTRAN 90 has two language elements that have object features: TYPE and MODULE. TYPE allows one to group data together but does not associate methods with the data.. One can declare an instance of a TYPE and access its data components. By itself, a TYPE does not provide encapsulation, since any section of code that defines a TYPE and declares an instance of it may directly access its data components. Encapsulation is achieved by placing the TYPE definition inside of a MODULE. FORTRAN 90 does allow one to build an abstract by combining TYPES and PROCEDURES (i.e. SUBROUTINES and FUNCTIONS) in a MODULE. The TYPES contain an internal data, which can be declared PRIVATE inside of a MODULE and thus encapsulated. The MODULE provides an interface via PROCEDURES that are PUBLIC and operates on the contained TYPE. Generic components could be implemented either by class or templates. Constructing Generic objects using template has an advantage that their instantiation is done at compiler time and therefore avoid run time overheads associated with the late-binding mechanisms in different programming languages.

Deliverable: Transformed code in object-oriented paradigm.

Figure 4 shows the transformation of legacy system into C++, an object-oriented programming language.

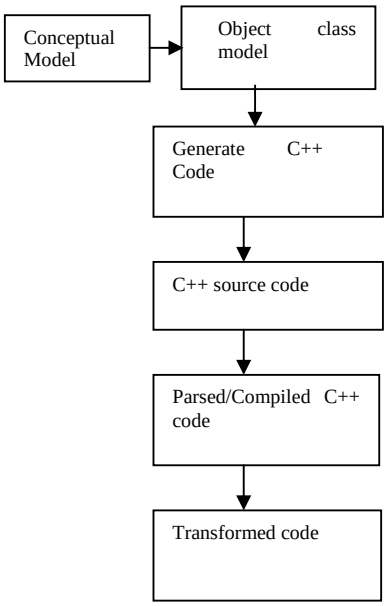


Figure 4: Transformation of legacy system into C++

4. Discussion and Future Work

Legacy system modernization should be effective and semantic preserving. There have been several other approaches developed for legacy system modernization as discussed in section 2. Our approach is primarily distinguished by the use of software architecture reconstruction tool and reusing the legacy code making it more maintainable and evolvable. Our approach emphasises on the modernization of legacy system through redevelopment approach so that the continuos modernization is possible. We believe that reusing software is very important for software modernization. From the

economic prospective, Graham (1991) reported that reuse strategy could save more than 20% of the development cost. We have to accept the fact that software takes years to mature and therefore it is important that we do not discard the useful software, which were written in procedural programming languages. We provide a legacy software modernization technique where design structures and function dependencies can be extracted from source code, visualized, and restructured. Bad broken legacy code is recovered, restructured and is reused for the modernization of the system. This strategy can be applied to legacy system that needs to be transformed to object-oriented paradigm for reduced cost maintenance and reusability. The modernization process gives a step-by-step procedure for optimising improvements in cohesion and coupling using design patterns and generic programming. Also, the process of modernization would improve an analyst's understanding of the legacy system and the code reuse.

What we are trying to accomplish from our research is to increase the overall comprehensibility of the legacy software for maintenance and future evolution. It will facilitate legacy code reuse, which has been a significant issue in the movement towards software reusability. Legacy software would be reconstructed for better understanding and to handle the sheer volume and complexities associated with it. When all the phases of modernisation are complete, the code is easy to follow at high level, the architecture is transparent and how each component of the architecture achieves its function is transparent. It will allow reusability and better maintainability of the code.

Currently we are verifying this approach using the Automatic Cane Railway Scheduling System (ACRSS), developed in 1987. ACRSS uses data describing the cane railway layout, the harvesting pattern of the relevant growers and some operational parameters to produce a schedule. It has 150 KLOC. The codes are being written in FORTRAN 77. These codes have not yet been modernized. Our on going work involves identifying object-oriented features from the procedural ACRSS software and modernising it.

5. References

1. Weiderman NH, Bergy JK, Smith DB, Tilley SR. Approaches to Legacy System Evolution. Report CMU/SEI-97-TR-014, December 1997, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213.
2. Comella-Dorda S, Wallnau K, Seacord RC, Robert J. A survey of Legacy System Modernization Approaches. Report CMU/SEI-2000-TN-003, April 2000, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213.
3. Seacord RC, Plakosh D, Lewis GA. Modernizing Legacy Systems, Addison-Wesley Professional; 1 edition, 2001.
4. Bass L, Clement P, Kazman R. Software Architecture in Practice, Addison-Wesley Professional; 1st edition 2002
5. O'Brien L, Tamarree V. "Architecture Reconstruction of J2EE Applications: Generating Views from the Module Viewtype," Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-2003-TN-028, November 2003.
6. Kazman R, Carriere S J. Playing Detective: Reconstructing Software Architecture from Available Evidence, *Journal of Automated Software Engineering*, pp 107-138, April 1999.
7. The Dali Architecture Reconstruction Workbench: http://www.sei.cmu.edu/ata/products_services/dali.html
8. Hainaut J L. Database Reverse Engineering. *Doctoral Dissertation*, University of Namur- Institute d'Informatique, 211B-5000 Namur, Belgium, 1998.
9. Bowman T, Holt R C, Brewster N V. Linux as a Case Study: Its Extracted Software Architecture. *Proceedings of the International Conference on software Engineering*, Los Angeles, May 1999.
10. Plakosh, Daniel; Hissam, Scott; and Wallnau, Kurt. Into the Black Box: A case study in Obtaining Visibility into Commercial Software *(CMU/SEI-99-TN-010)*. Pittsburg, Software Engineering Institute, Carnegie Mellon University, 1999.
11. Stoermer C, O'Brien L, Verhoef C."Practice Patterns for Architecture Reconstruction" *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*, 29 October-1 November 2002 at Richmond, Virginia, USA.
12. Kang B K, Bieman J (1998) 'Using design abstractions to visualize, quantify, and restructure software', *The Journal of Systems and Software*. 42(2), 175-187.
13. Decyk V K, Norton C D. "Modernizing FORTRAN 77 Legacy Codes", *Conference on Computational Physics 2000 (CCP 2000)*, Advance Program and Technical Digest, Gold Coast Queensland, Australia, pg 104, December 3-8, 2000.
14. Muller, H A, Mehmet, O A, Tilley S R, &Uhl, J S, "A Reverse Engineering Approach to System Identification." *Journal of Software Maintenance: Research and Practice* 5, 4 (December 1993): 181-204.
15. Deursen A, Klint P, Verhoef C. Research Issues in the Renovation of Legacy Systems. Proceedings of the Fundamental Approaches to Software Engineering, LNCS, Springer Verlag, 1999

Supporting Static and Dynamic Feature Modeling by Formal Concept Analysis

Jian Kang and Hongji Yang
Software Technology Research Laboratory
De Montfort University, Leicester, LE1 9BH, England
{jkang, hyang@dmu.ac.uk}

Abstract

Architectural recovery is an important task within reengineering projects. And feature modelling is introduced as an additional step in a system's architecture recovery process. However, the presentation and analysis of the feature models are still largely informal. There is also an increasing need for methods and tools that can support automated feature analysis. Nevertheless, techniques shall combine dynamic and static analysis to rapidly focus on the system's architecture. This paper presents a formal restructuring approach to the specification and verification of feature model.

Keywords*: Software Architecture, Reengineering, Feature Modelling, Static and Dynamic Analysis, Architecture Reconstruction, Formal Concept Analysis (FCA).*

1. Introduction

[9] Architectural recovery stats the basis for a new system architecture. The activity aims to recover information about the current system architecture, which was lost or become outdated over years. In the case of little and outdated system documentation the most reliable information can be derived from source code. However the lack of abstract information hinders the comprehension. There is often an abstraction gap between source code, system documentation and architecture. To bridge this gap domain information is applied with great success, i.e. in form of the feature models [10].

Research on feature modelling has revived much attention in the domain engineering community. Feature-Oriented Reuse Method (FORM) [3] and Feature-Oriented Domain Analysis (FODA) [2] are domain engineering methods that concentrate on modelling and analyzing a product line's commonalties and variabilities in terms of features. In the first paper on feature modelling in 1990 [2], Kang et al. suggested the formalization of the features as a future direction. However, since then formalization of feature models has not been taken up. In this paper we present an approach to combines dynamic and static analyses to rapidly focus on the system's parts urgently required for a goal-directed process of architectural recovery.

The rest of this paper is organised as follows. Section 2 presents feature modelling; Section 3 introduces Formal Concept Analysis (FCA), which is the key techniques used in our proposed approach; Section 4 describes the overall dynamic and static analyzing process we have developed

2. Feature Modelling

Conceptual relationships among features can be expressed by feature model as proposed by Kang et al. [2] A feature model consists of a feature diagram and other associated information (such as rationale, constraints and dependency rules). A feature diagram provides a graphical tree-like notation that shows the hierarchical organization of features. The root of the tree represents a concept node. All other nodes represent different type of features.

Features represent distinguishable characteristics of a concept. A concept consists of a set of related features with constraints. In [1], Features and Concept is given the definitions as that concept is a special kind of feature, which is represented as a subset of Feature.

3. Formal Concept Analysis

Concept analysis is a mathematical technique that provides insights into binary relations. This mathematical foundation of concept analysis was laid by Birkhoff in 1940. Primarily Snelting has recently introduced concept analysis to software engineering. Since then it has been used to evaluate class hierarchies [6], explore configuration structures of pre-processor statements [4], [6], for redocumentation [5], and to recover components [8].

The binary relation in our specific application of concept analysis to derive the scenario-subsystem relationships states which subsystems are required when a feature is invoked.

Concept analysis is based on a relation R between a set of object O and a set of attributes A, hence $R \subseteq O \times A$.

C = (O, A, R) is called formal context. For a set of common attributes σ and common objects τ

$$X = \{o \in O \mid \forall a \in Y: (o,a) \in R\},$$
$$Y = \{a \in A \mid \forall o \in Y: (o,a) \in R\}.$$

In section 4.1 the formal context for applying concept analysis to derive the scenario-subsystem relationships will be laid down as follows;

- subsystems will be considered objects,
- scenarios will be considered attributes,
- a pair (subsystem s, scenarios S) is in relation R if s is executed when S is performed.

4. Dynamic and Static Analysis

In order to derive the feature modelling via concept analysis, one has to define the formal context (objects, attributes, relation) and to interpret the resulting concept lattice accordingly.

4.1 Dynamic Analysis

The goal of dynamic analysis is to find out which subsystems contribute to a given set of features. For each feature, a scenario is prepared that exploits this feature. Hence, subsystems will be considered objects of the formal context, whereas scenarios will be considered attributes. In the reverse case, the concept lattice is simply inverted but the derived information will be the same.

The relation for the formal context necessary for concept analysis is thus defined as follows:

(s, S) $\in$ R if and only if subsystem s is required for scenarios S; a sub system is required when it needs to be executed.

In order to obtain the relation, a set of scenarios need to be prepare where each scenario executes preferably only one relevant feature. Then the system is used according to the set of scenarios, once at a time, and the execution summaries are recorded. Each system run yields all required subsystems for a single scenario, i.e., one column of the relation table can be filled per system run. Applying all scenarios provide the complete relation table.

4.2 Features and Scenarios

Because one feature can be invoked by many scenarios and one scenario can invoke several features, there is not always a strict correspondence between features and scenarios. Figure 1 is an abstracted class hierarchy displayed as a concept lattice drawn by ToscanaJ-1.5.1, which is a formal concept analysis tool we adopt in our case study. And we will analysis sufficient the relationships information containing in the lattice in case study section.

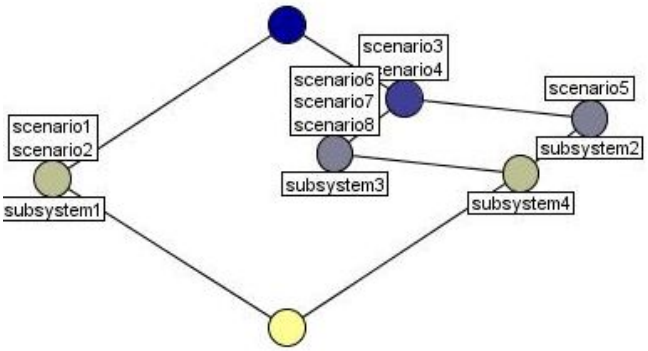


Figure 1. Abstracted class hierarchy displayed as a concept lattice

4.3 Static Analysis

In this static analysis process, subsystems and scenarios are detected and separated. The static relationships between these components are recovered. Also from the concept lattice, we can easily derive all subsystems executed for any set of relevant features which are synonyms. However, this gives us only a set of subsystems, but it is not clear which of them are general-purpose subsystems that are only used as building blocks for other components but do not contain any feature-special logic. We also will further this discuss in our case study section.

5 Summary

We showed how analyses performed over feature models via dynamic and static formal concept analyzing unbalanced architecture and design redundancies. Therefore reengineers can directly find out how features and feature modelling in overcoming system architecture decay and obsolescence of the whole system.

References

[1] J. Sun, H. Zhang, et. Formal Semantics and Verification for Feature Modelling, In *Processings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECC'05)*, 2005.

[2] K. C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, November, 1990.

[3] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain Reference Architectures. *Annals of Software Engineering*, 5: 143-168, 1998.

[4] M. Krone, G. Snelting, On the inference of Configuration Structures From Scource Code, In *Processings of the International Conference on Software Engineering*, pp. 49-57, May 1994.

[5] T. Kuipers, L. Moonen, Types and Concept Analysis for Legacy Systems, In *Processings of International Workshop in Program Comprehension*, 2001.

[6] I. Pashov and M. Riebisch. Using feature modelling for program comprehension and software architecture recovery. In *Processings of the 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBC' 03)*, Huntsville Alabama, USA, IEEE Computer Society, April 2003.

[7] I. Pashov, M. Riebisch, I. Philippow, Supporting Architectural Restructing by Analyzing Feature Models, In *Processings of the 8th European Conference on Software Maintenance and Reengineering (CSMR' 04)*, 2004.

[8] G. Snelting, Reengineering of Configuration Based on Mathmetical Concept Analysis, *ACM Transactions on Software Engineering and Methodology* vol. 5, no. 2, pp. 146-189, April 1997.

[9] G. Snelting, F. Tip, Reengineering Class Hierarchies Using Concept Analysis, In *Processings of the ACM SIGSOFT Symposium on the Foundations of software Engineering*, pp.99-110, November 1994.

[10] A. Van Deursen, T. Kuipers, Identifying Objects Using Cluster and Concept analysis, In *Processings of International conference on Software Engineering*, 1999.

Recovering General Layering and Subsystems in Dependency Graphs

Andrew J. Malton
School of Computer Science
University of Waterloo
Waterloo, Canada
ajmalton@uwaterloo.ca

Abstract

An essential steop in re-use projects is the construction of a suitable tailored view of the architecture of the subject software. This position paper describes an approach to recovering software architecture in layers and modules, using matehmatical programming.

1. Introduction

If we are to identify and mine out reusable material from software, we have to identify and understand the dependencies between parts. Since the notion of dependency between parts is not easily defined, let us take the reuse motivation as definitive for the moment: A is *dependent* on B exactly when reusing A entails reusing B. If we want to copy A into a new setting to do its work there, we find we have to drag B along for the ride.

In a huge system of interdependent pieces of software, whose boundaries (procedures, methods, source files, classes, packages, etc.) are defined by programming language semantics, operating system rules, and design conventions, we are faced with the huge task of discovering and assessing their dependencies and (for the sake of reuse) applying architectural transformations to decouple the parts that are to be copied out and reused.

The architectural pattern of *layering* [B] is commonly presented as a design choice, when re-use is expected. The idea [Figure 1] is that observing the characteristic *layering discipline*, according to which inter-layer dependency is "down only", results in a design in which individual layers can more easily be replaced or reused.

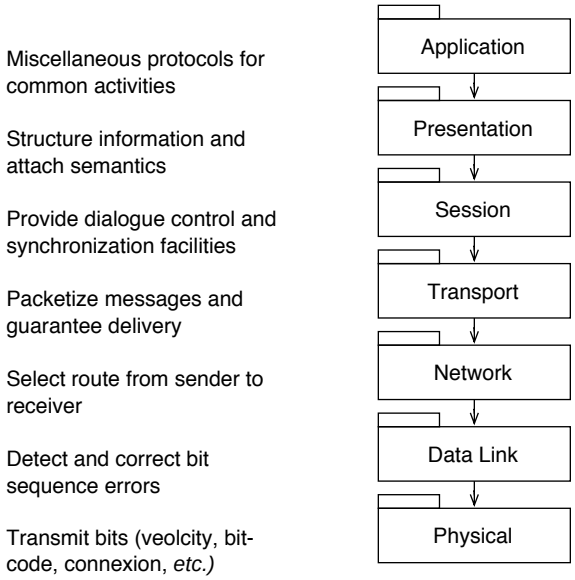


Figure 1. OSI Reference Layering

In addition, however, the layering pattern aids *understanding*, because a developer on a software project can concentrate his expertise on the design concepts of the system *at a given layer*: he ignores the layers above which his layer doesn't depend on; and he needs only a user's knowledge of the layers below, which he views as a provider of services. This would be particularly true when the layering discipline is strict, that is, inter-layer dependency goes down one layer only. The layering provides a mental model [Holt] of the software architecture.

This note sketches a technique for recoving layers using mathematical programming.

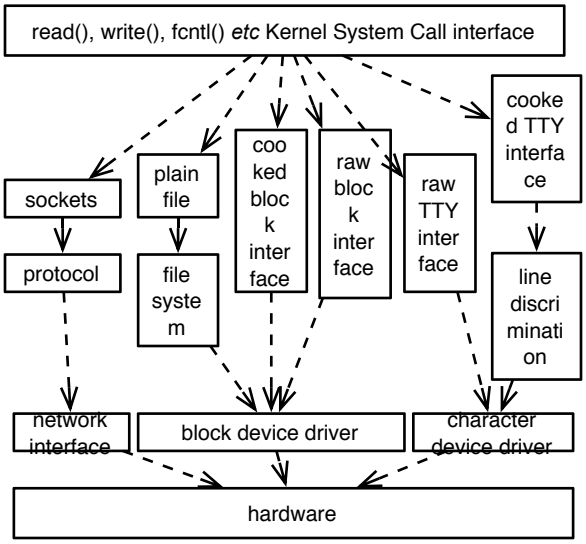


Figure 2. Unix kernel "toaster" layering

2. Layering in Large Systems

A colleague's chance remark to the effect that a certain huge (i.e. □ 107 LOC) system under study exhibited "the usual large-scale layering" led me to the observation that as a large system grows and evolves, the historical processes plus the need for local understanding per the previous paragraph tends to give rise to a *general layering* architecture which transcends, or cross-cuts, any other architecture patterns and plans. This layering will be seen throughout the system.

A familar toaster-style view [Figure 2] of Unix exhibits general layering. A view [Figure 3] of Java applications in OS/X exhibits general layering to a deeper degree. What is happening here? As a system evolves, grows, and adapts to become more open to variations, plug-ins, third-party extensions, and the like, it seems to organize itself into layers – and those layers affect the design and evolution going forward. A new part, or new development work, will tend to be "in a certain layer"

3. Subsystems and Modules

The familiar reverse-engineering plan-of-attack (e.g. in clustering analysis [Mancoridis, Tzerpos, etc.] or in static visualization [Shrimp, lsedit, GuPRO, etc..]) is to discover, and possibly synthesize, a modular view of a given system, somehow presenting its abstractions as subsystems, containing modules, with various kinds of explicit or implicit dependency. Explicit dependencies are those visible directly from the code. Implicit dependencies may be discovered by fact-base analysis or by introducing additional facts from documentation or monitored or instrumented execution. Typically, however, a modular view (approximately Kruchten's "Development View" [Kruchten]) is the goal.

However, the interaction between layering and modularity is not clear. Many authors (e.g. Busch, Johnson, Shaw) will prefer the view that layering is an architectural pattern guiding the design of a subsystem. See the subsystem; or open it up and see the layers. Other authors (e.g. Larman [Larman]) teach that a whole subsystem inhabits a given layer, so that the layering is a macro-level plan. This seems in keeping with actual practice. Probably the best view is that layering has two meanings: tactically, as a design pattern; and strategically or emerging from evolutionary pressure, as a macro-level phenomenon.

In this research I tried to take a middle view, albeit preferring the macro-level account. It seems that the general layering of a system colours the design of all its subsystems. See the subsystem, open it up and see the familiar layering. See another subsystem, open it up and see the same layering again. The general layering and the

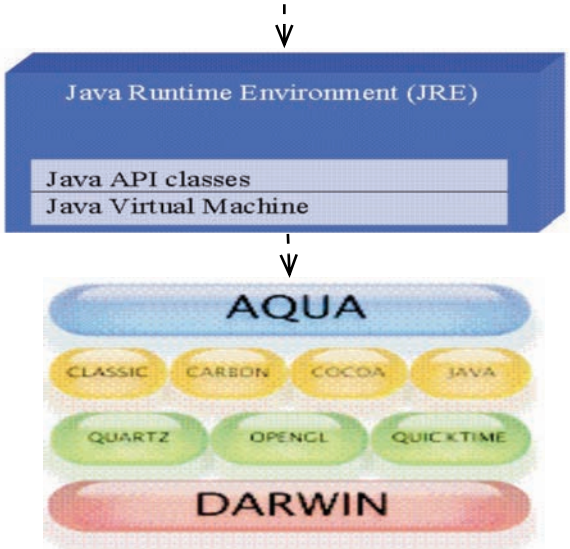


Figure 3. Java application layering.
Original Images © Sun, Reilly, Apple.

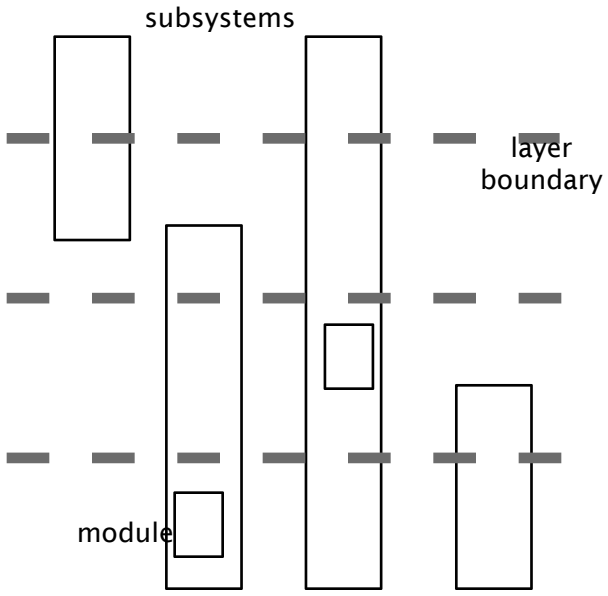


Figure 5. Layers, Subsystems, and Modules

subsystem structure cross-cut each other. This view is also found in the literature; in fact it is a very old view. well expounded by Parnas [Parnas] for example.

In this view a subsystem inhabits several layers, although possibly not all of them. Some developers might have an expert's view of a layer (as before) – and others may now have an expert's view of a subsystem. Reuse of a subsystem will be easiest when shoe-horning it into a new context where the general layering pattern is similar. Replacement of a layer (which is also reuse) will be easiest when slicing into a system with a similar modular architecture.

Subsystem design discipline is based upon the familiar modularity triad of cohesion, coupling, and information hiding. This does not change when we take general layering into account. However, the general layering discipline and the modularity discipline affect each other. If adjacent layers determine different "levels of abstraction" and adjacent subsystems determine different "conceptual concerns" then one way to improve cohesion is to avoid crossing both subsystem and level boundaies in the same dependency. In summary, in the ideal case:

- The general system layering imposes an internal layering on every large subsystem.
- Dependencies between parts are in the same subsystem or in the same layer.
- Equivalently, a dependency on a lower layer belongs to the same subsystem.

From these rules it folllows that a part of the design that belongs to a given layer and subsystem is of interest, due to the constraints its dependencies. Let's call it a *module*. [Figure 5]

4. Architectural Design Recovery

The above discussion leads to a technique for design recovery that complements clustering. For this work, suppose a dependency graph is given, extracted by the usual means from available artefacts (such as source code). There are evidently many possibly partitions which observe the "down only" rule, thereby qualifying as layerings. The diagram [Figures 6a,b] show a couple of examples.

By expressing the layering design as an mathematical program (i.e. and optimization problem) we can investigate which are the "best" recovered layerings. In an optimization problem, we describe the data arithmetically and give a function to maximize or minimize as the desired solution.

A dependency graph is described arithmetically, in AMPL [Fourer], by a set of Dependencies, which are pairs of nodes. A layering solution is a boolean function In[x,y], which is 1 if x is in layer y, and 0 otherwise. The constraints which make this a layering are shown in the

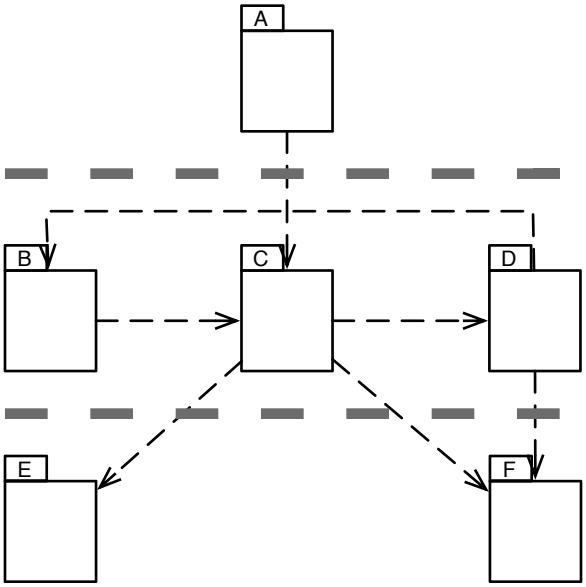


Figure 6a. A Possible Layering

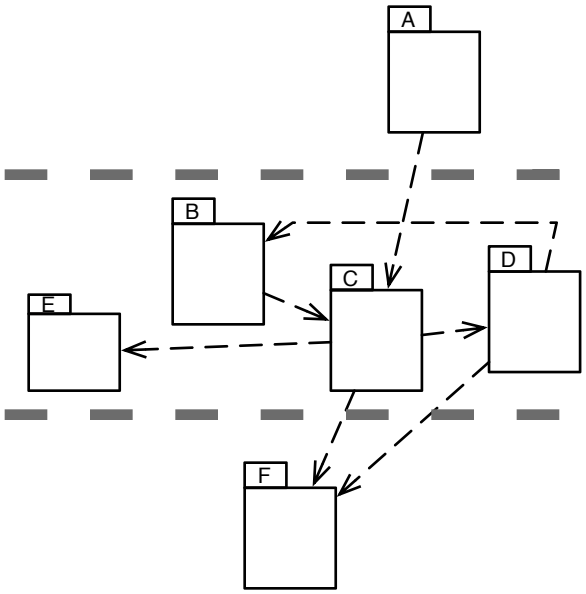


Figure 6b. A Different Layering

diagram. [Figure 7] I don't know a way to include the number of layers in the model, nor (consequently) to maximize the number of layers if that were desirable. Instead, we solve for other constraints while fixing the number of layers.

The optimization expression dictates which model of layering seems most effective. The diagram [Figure 8] shows several possibilities. Oddly, a good one seems to be to maximize the number of layer-crossing dependencies (by minimizing the number of same-layer dependencies). This has the effect of making the layers "about the same size" since to maximize the number of crossings there should be lots of nodes in each layer. It has the cognitive value, that with many crossing dependencies, the "essential" structure of a given layer is shown. When focusing attention on a given layer, dependencies into and out of it can be ignored.

It also appears that allowing, in general, that dependencies should cross more than one layering boundary is pointless: the optimisation then merely adds layers until the structure resembles one which would have appeared anyway with fewer layers and fewer "deep" dependencies. Perhaps the number of deep dependencies can be minimized: further work is needed here.

Having recovered a layering, a modular decomposition follows from the rules: every dependency which crosses a layer boundary puts the two modules in the same subsystem. This was implemented using **grok** [grok] and works nicely for theoretical cases. However, because the ideal rule is not always followed in practice, modularity recovery will have to be be combined with clustering or code structure facts, and the ideal rule interpreted more loosely. There is scope here for both further optimization, and for discovered architectural "repairs" prior to re-use or preventive maintenance.

```
# Constraints

# No layer is allowed to be empty.
subject to layer_not_empty {l in Layers}:
    sum{s in Entities} In[s,l] >= 1;

# Every entity must be at a unique layer.
subject to in_unique_layer {s in Entities}:
    sum{l in Layers} In[s,l] = 1;

# Dependencies must be down, and not too deep, in the layer hierarchy.
subject to down{(x,y) in Dependencies, l in Layers}:
    In[x,l] - sum{i in 0 .. min(l,MAXIMUM_DEPTH)} In[y,l-i] <= 0;

# Permission to depend at the same layer must be granted
subject to permission{(x,y) in Dependencies, l in Layers}:
    In[x,l] + In[y,l] - P[x,y] <= 1;
```

Figure 7. Layering constraints in AMPL

```
# Objective: as few same-layer dependencies as possible.
minimize call_weight:
    sum {(x,y) in Dependencies} P[x,y];

# Objective: maximize outer layers:
maximize outer_layer_size:
    sum{s in Entities} (In[s,TOP] + In[s,BOTTOM]);

# Objective: maximize weight:
maximize node_height:
    sum{s in Entities, l in Layers} l * In[s,l];
```

Figure 8. Layering optimization functions in AMPL

5. Anchors and Façades

The ideal rule for modular layering, as above, means that modules at the "top" of a subsystem and modules at the "bottom" are very important. [Figure 9]

Modules at the top, which tend to appear in an upper layer, have the purpose of "summarizing" the functionality of a subsystem: subsystems at the same or higher level can use those "top" modules, and thereby gain mediated access to the deeper levels of the other subsystem. We term such "top" modules as *façades* in keeping with the O-O pattern language.

Modules at the bottom, which tend to appear in a lower layer, have the purpose of providing access for a subsystem to facilities at a level "below" the subsystem's natural level. For example, a subsystem S which implements a conceptual data domain and a user interface for it, still needs access to a general database facility. Rather than to cross into the technical services layer at the same time as crossing into the database subsystem, the rule suggests introducing a "low level" module into S whose sole job is providing S-tailored persistence features by dependency on low-level database facilities. Such a module is an *anchor* because it anchors S in the lowest level where it needs support.

8. Conclusion

By recovering the archtiecture of existing systems according to the accepted principles of modularity, researchers have hoped to discover reusable modules.

By adding the natural general layering to the mix, I am investigating the possibility of improving the understandability of recovered architectures and reusability of recovered subsystem and module structures.

This work also shows the possibility of using linear optimization techniques to explore the space of recovered designs.

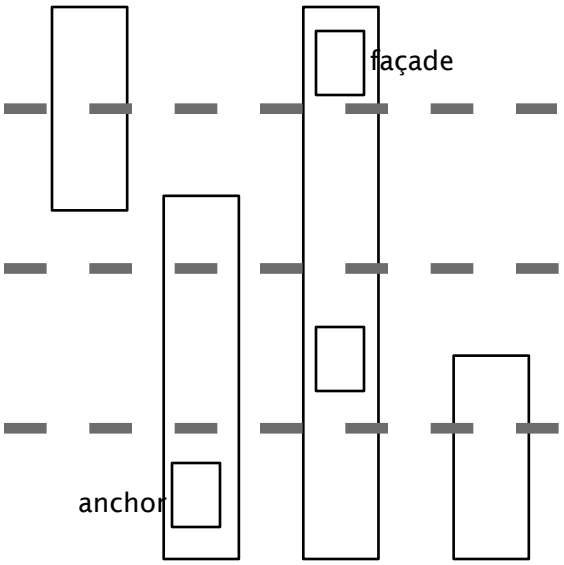


Figure 9. Façades and Anchors

References

[Fourer] R. Fourer et al. AMPL: A Modeling Language for Mathematical Programming. Cole, 2002.

[Buschmann] F. Buschmann et al. Pattern-Oriented Software Architecture, Volume 1, A System of Patterns. Wiley, 1996.

[HC] R.C. Holt and J.R. Cordy. The Turing Plus Report, CSRI, U Toronto, Feb 1987

[H] R. C. Holt. "Software architecture as a shared mental model", IWPC 2002.

[Mancoridis] S. Mancoridis et al. "Bunch: A clustering tool for the recovery and maintenance of software system structures", ICSM 1999.

[Tzerpos] P. Andritsos and V. Tzerpos. "Software clustering based on information loss minimization", WCRE 2003

[Kruchten] P. Kruchten. "The 4+1 view model of architecture," IEEE Software, 12 (6), 1995.

[Shrimp] Storey, M.-A, et al. "Cognitive design elements to support the construction of a mental model during software exploration," JSS 44, 1999

[lsedit] P. Finnigan et mult. al. "The Software Bookshelf". IBM SJ 36.4, 11/1997.

[GuPro] J. Ebert et al., eds. "GUPRO—Generische Umgebung zum Programmverstehen", Fölbach, 1998. see http://www.uni-koblenz.de/FB4/Contrib/GUPRO/Site/Home

[Larman] C. J. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Prentice Hall, 2004.

[Parnas] Parnas, DL, "On a 'Buzzword': Hierarchical Structure", Proc IFIP '74

[grok] R. C. Holt, Introduction to the Grok Programming Language, Ric Holt, May, 2002. See http://plg.uwaterloo.ca/~holt/papers/grok-intro.doc

Supporting Migration to Services using Software Architecture Reconstruction

Liam O'Brien, Dennis Smith, Grace Lewis
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213 USA
+1 412 268 7727
{lob, dbs, glewis}@sei.cmu.edu

ABSTRACT
There are many good reasons why organizations should perform software architecture reconstructions. However, few organizations are willing to pay for the effort. Software architecture reconstruction must be viewed not as an effort on its own but as a contribution in a broader technical context, such as the streamlining of products into a product line or the modernization of systems that hit their architectural borders. In this paper we propose the use of architecture reconstruction to support System Modernization through the identification and reuse of legacy components as services in a Service-Oriented Architecture (SOA). A case study showing how architecture reconstruction was used on a system to support an organization's decisions making is presented.

Keywords
Architecture, Architecture Reconstruction, Service Oriented Architecture, Migration to Services, System Modernization.

1 INTRODUCTION

Much research has been done in software architecture reconstruction in the past several years [2,4,6,7,8,10,13,14,16,18,19] and many techniques and methods have been developed along with tools to support them [3,5,11,17]. Software architecture reconstruction has been used to:

- (Re)Document the architecture of existing systems to improve the understanding of the architecture
- Check the conformance of an as-designed architecture with the as-built architecture and remove unexpected dependencies
- Trace architecture elements to the source code, for instance to measure the impact of architectural changes

Stoermer, et al. [22] outlined a set of application contexts where architecture reconstruction has been and can be used. In this paper we focus on how architecture reconstruction can be used to support System Modernization through the identification and reuse of legacy components as services in a Service-Oriented Architecture (SOA).

There has been much work on moving existing legacy functionality to be used as web services or to web environments. Sneed and Sneed [21] outline an approach for creating web services from legacy host programs. Kontogiannis and Zhou [12] outline an approach to migrating legacy applications through identification of major legacy components and migrating these procedural components to an object-oriented design, specifying the interfaces, automatically generating the wrappers and seamlessly interoperating them via HTTP based on SOAP messaging. Litoiu [15] outlines issues such as performance and scalability related to migration of legacy applications to web services.

If an organization decides to move toward the use of services and an SOA to take advantage of the benefits that can be gained by using a service-based approach, there are several obstacles that it may have to overcome:

- Usually legacy systems are not well understood or documented,
- The dependencies between the components that can be migrated to services may not be known or documented
- There is a need to determine the feasibility of whether or not to invest in doing the migration.

This paper outlines how architecture reconstruction can be used to overcome some of these obstacles and to support the organization's decision making process. Using architecture reconstruction can help to get a better understanding of legacy applications and identify and document component dependencies that may not have been documented or known. This information can be used to better inform the decision making process.

For a component to be migrated to a service the component should be self-contained and loosely coupled. If a component has a lot of dependencies especially functional dependencies where the component is calling other functionality outside of it, then this would reveal that the

component is not self- contained. If there are many calls or other dependencies from outside the component to the component then it may highlight that the component is tightly coupled to other parts of the system whereas it should be loosely coupled if it is to become a service.

This paper focuses on the use of architecture reconstruction as a decision-making tool. We have applied this approach with an organization that wanted to get a better understanding of the feasibility of migrating and reusing several of its legacy components as services in an SOA. To get a better understanding of the legacy system we used the ARMIN tool to analyze the legacy system and we will report on the outcomes of that analysis.

The remainder of the paper is organized as follows. Section 2 gives a brief overview of architecture reconstruction and the ARMIN tool. Section 3 outlines how architecture reconstruction can be used to identify dependencies between components in a system and produce better documentation. Section 4 outlines the case study in applying architecture reconstruction for better decision making on migration to an SOA. Section 5 presents conclusions and outlines some future work.

2 ARCHITECTURE RECONSTRCUTION

Architecture reconstruction is the process of reconstructing or recovering the architecture of an implemented system. The SEI has developed tools to support this process and the current tool is the Architecture Reconstruction and MINing (ARMIN) tool. ARMIN uses information that is extracted from the source code in the form of an RSF (Rigi Standard Format) file. An RSF file has a set of elements and relations between these elements. The process that is used to reconstruct views of the architecture is outlined in Figure 1.

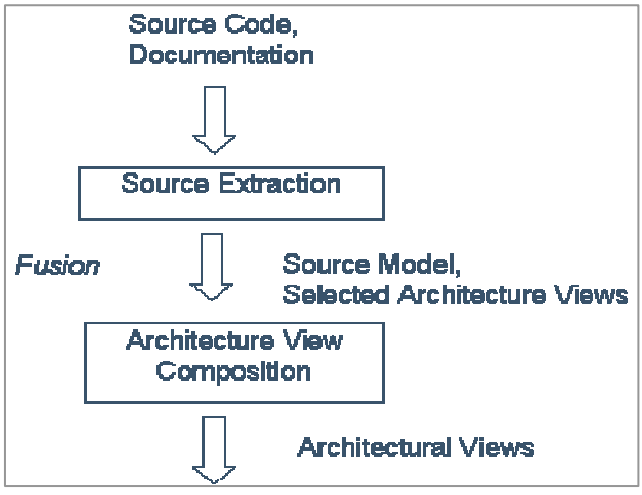


Figure 1: Architecture Reconstruction Process

To extract the information from the source code we use commercially available tools such as the Understand toolset from Scientific Toolworks Inc. [20] or the Imagix 4D tools from Imagix Corporation [9]. After the data is extracted, it is imported into ARMIN.

The ARMIN tool has several major components:

- Navigator – for defining a project which includes the elements and relations that make up the source model and loading and browsing the source model from the RSF file.
- Aggregator – for generating the views of the models generated.
- Interpreter – for creating the abstractions and models using ARMIN's Architecture Reconstruction Language (ARL).
- Repository – where data loaded into a project can be stored.

ARMIN has been used to analyze systems written in C, C++, Fortran and Java and in various combinations of these languages. It has been used on systems up to 5 million lines of code.

As the reconstructor follows the reconstruction process using ARMIN, various abstractions are built using ARMIN's ARL command script to aggregate information and produce higher-levels models and views of that information. Abstractions could include hiding functions inside of file or aggregating classes or files into components. This process is continued until a set of views that are needed are produced. For example if a Layer view of the system is required a typical set of abstractions may include hiding functions within classes, aggregating classes to components and aggregating components to layers. A typical Layer view generated by ARMIN is shown in Figure 2. This has three layers; Application, Bootloader and Communication.

3 IDENTIFYING DEPENDENCIES BETWEEN COMPONENTS FOR SERVICE MIGRATION

In order to migrate legacy components to services in an SOA it is important that the dependencies for a legacy component are identified and understood. There can be various types of dependencies between a component and the rest of the system. Dependencies include:

- Functional dependencies where the component uses other parts of the system in order to carry out its functionality or other parts of the system use the components.
- Data dependencies where global data is shared

between the component and other parts of the system.

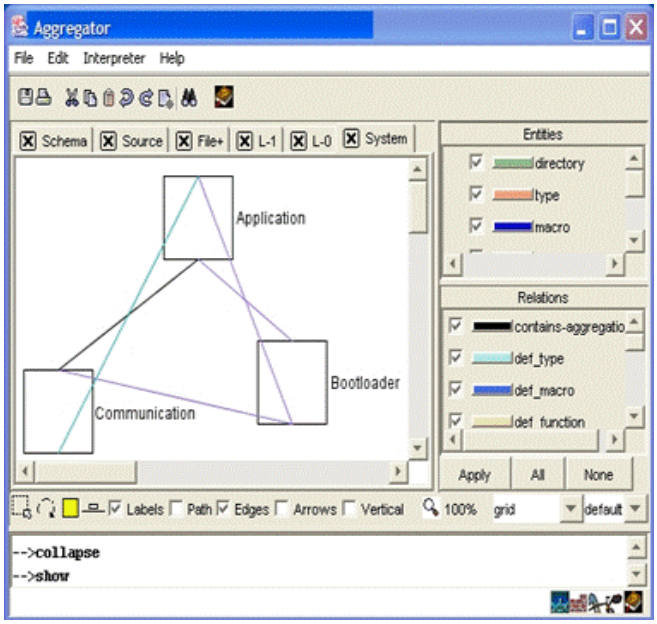


Figure 2: Layer View Generated by ARMIN

In order to help identify these dependencies we can extract the detailed information from the source code that will help us to build abstractions that highlight these dependencies. We can extract information about the elements (such as the functions, files or classes) that are contained in the system and additional elements such as directories, if the system is decomposed on a file system. These will be used to build the abstractions to identify the components.

Along with the elements we also extract a set of relations between these elements. Specific relations between the elements that highlight dependencies include:

- Calls between functions – what functions call each other – *calls function function*
- Data definitions – where are global variable defined – *defines_var file variable*
- Data usage – what functions use global variables (both references and assignments) – *uses_var function variable*

As well as the relations from the source code we can also generate additional relations such as:

- The relationship of files and directories – what files are contained within a directory – *contains_file directory file*
- The relationship of files or directories to subsystems or potentially components – what files or directories make up a subsystem or component – *in_comp component file* or *in_comp component directory*

The latter relations are useful if the system is decomposed in a file structure and the subsystems and components can be mapped to directories. Otherwise this information may be of little use in the reconstruction process.

Identifying Components

If the organization we are working with is already investigating how to migrate parts of the legacy system to services, as was the case with the organization in the case study, then they may already be familiar with what components have been identified as potential services and what elements make up those components (what group of classes or what group of file constitute a component). If not then we can begin to use various techniques to identify the components. There are different mechanisms for doing this including grouping, clustering and pattern recognition.

Once we have identified how we want to carry out the abstractions we can build a command script in ARMIN's ARL that will automatically build these abstractions. The ARL has built in operators that allow for gathering all elements within a relation into a multi-dimensional list, merging parts of the list, removing elements and relations from a particular view and generating visual representations of an underlying graph structure.

An example of a view of components for a system that is produced in ARMIN is shown in Figure 3. This view is from a reconstruction of the Duke's Bank System which is part of the EJB tutorial from Sun.

Identifying Dependencies between Components

During the reconstruction process the relations between the elements are aggregated into the higher level abstractions. For example a call between functions defined in different files becomes a dependency between files when the functions are aggregated to files. If a further aggregation of files to components is done, then the call becomes a dependency between components.

4

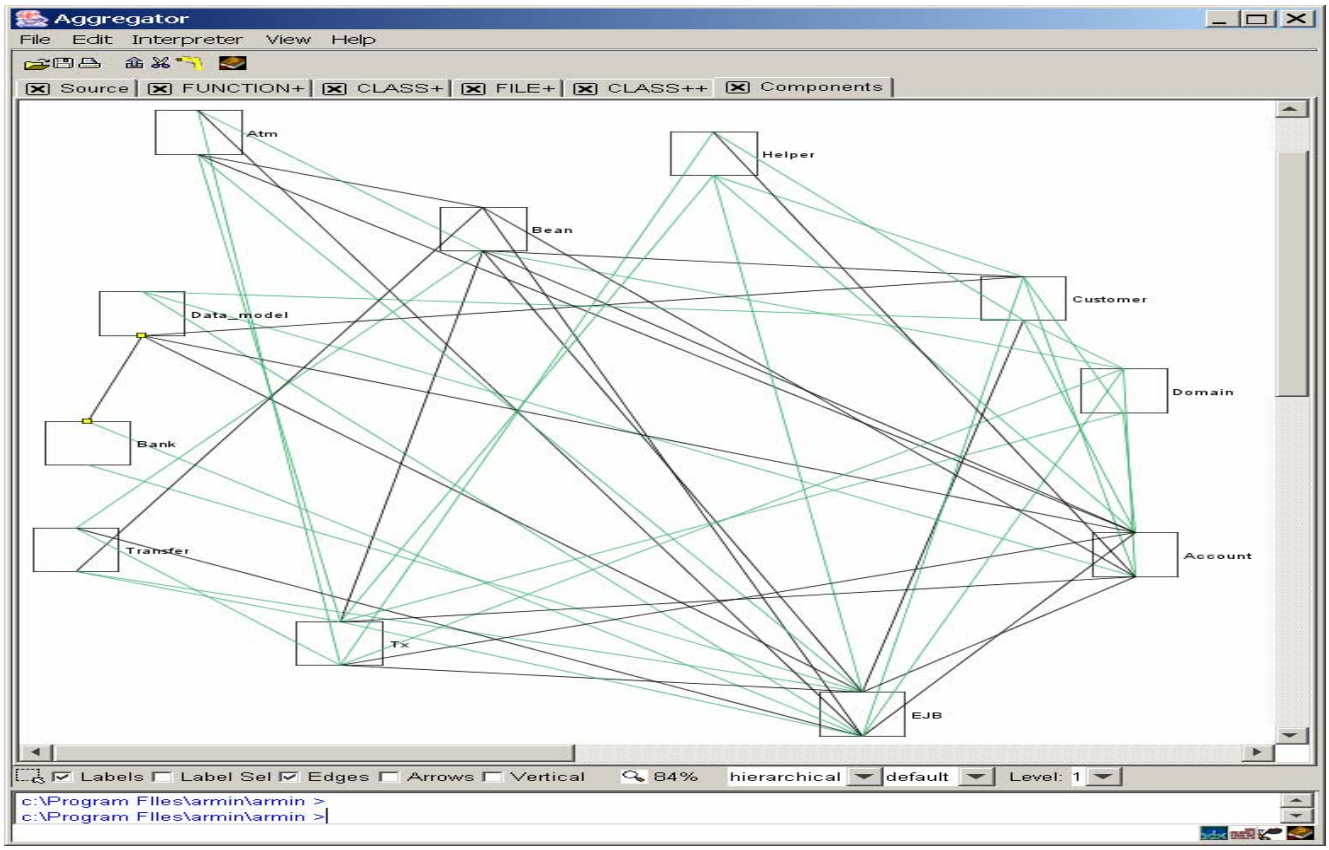


Figure 3: Component View produced by ARMIN

Identifying all of the external dependencies that a component has is important when considering migrating the component to a service. Of particular importance are dependencies between components that are candidate services, and dependencies between a component and the rest of the legacy system.

The dependencies that a component has, especially functional dependencies to other parts of the system, may mean that this component does not contain a self-contained piece of functionality. Depending on the number and type of dependencies from other parts of the system to the component, the component may be tightly coupled – a condition that goes against the principles for what a service should.

From the components view that is produced in ARMIN it is possible to select a particular component and show the dependencies that component has on the other components in the system. Figure 4 shows a view of the dependencies for the Account component (from Figure 3). The arrows show the direction of the dependency. For example the Atm component is dependent on Account but there is no dependency the other way.

It is possible to identify what exactly a dependency between components represents by selecting an edge connecting two components and showing the relations that make up the connections. For example the edge between the Account and Data_model components in Figure 4 identifies the dependencies between these components which are shown in Figure 5.

These dependencies include functional dependencies - in this case method calls between classes in the components, and additional dependencies between the classes. By examining the code the reason why these additional dependencies exist can be identified. In some cases these are caused because instance variables used in Data_model are of type AccountDetails or AccountController.

Using the architecture reconstruction techniques it is possible to identify the elements that make up the components that can potentially be migrated to services, and the dependencies between these components and the rest of the legacy system. We have applied this approach on a large system from a DoD organization. Details of this case study are outlined in the next section.

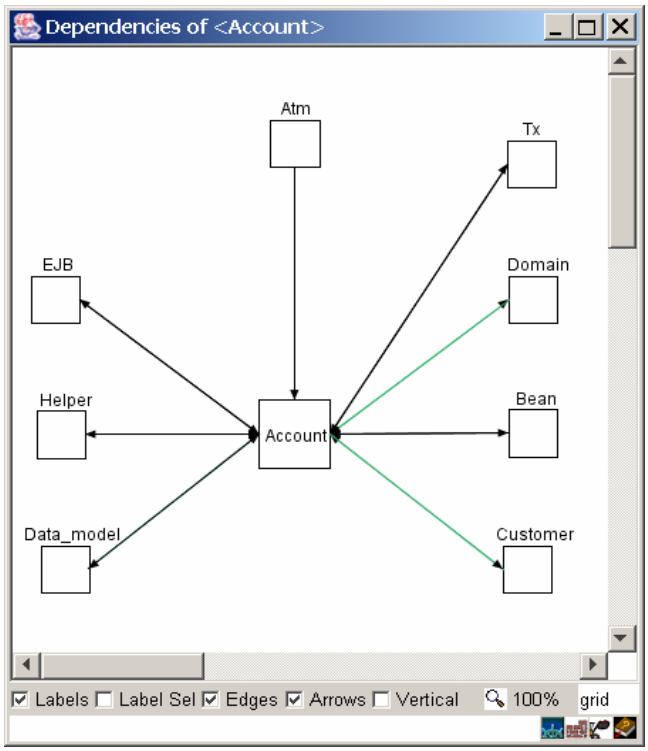


Figure 4: Dependencies for the Account Component

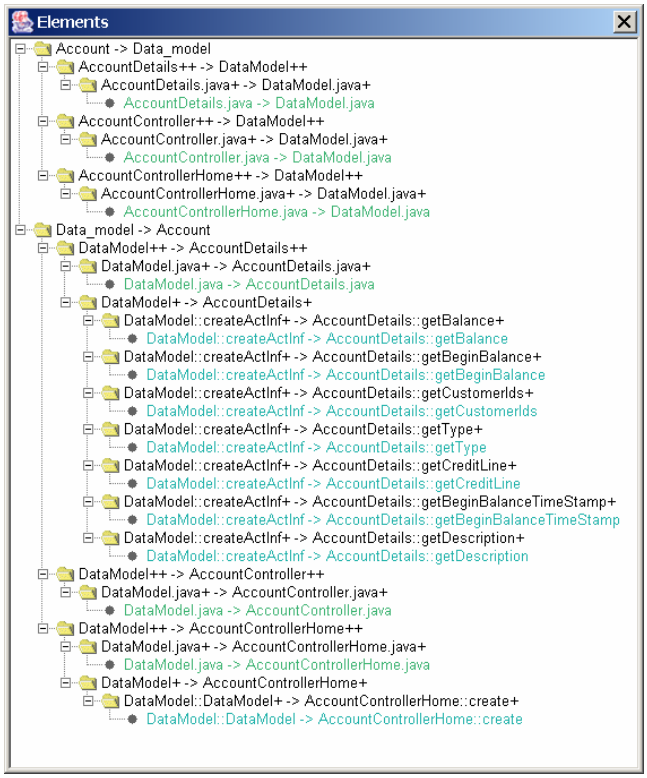


Figure 5: Dependencies between Account and Data_model

4 CASE STUDY IN SERVICE MIGRATION

In the case study existing components of a Command and Control (C2) system were being evaluated for their potential to become services in an SOA. The owners of the C2 system recognized that if a selected set of components from their system are converted to application domain services, they may have applicability for a broad variety of purposes. Our role was to perform a preliminary evaluation of the feasibility of converting a set of their components to application domain services within the SOA.

We initially met with the government owners of the system and the contractors who had developed the system. At this meeting we were given an overview of the legacy systems, the history of the systems, the migration plans, and the drivers for the migration. We were also provided with a brief orientation to the SOA. The SOA was developed so that C2 applications can be built as a set of interactions between infrastructure services (e.g., communication, discovery) and services that are specific to a domain (application domain services).

The system owner had done a preliminary identification of potential services that could be built from components of the legacy system. This analysis was derived from high level requirements for applications that were being targeted as users of services to be provided by the SOA. The system owner had matched legacy functionality to these high level requirements and provided some initial estimates of the contents of the potential services.

Target SOA

We investigated the target SOA through an analysis of available documentation and through a meeting with the developers. The target SOA is currently under development. It is being built using a variety of commercial products and standards, along with significant custom code. The effort is focused on satisfying a number of specific quality attributes important to the DoD, such as performance, security, and availability. In order to meet these needs, the SOA will impose a number of constraints on potential services. Because the SOA is still under development, the specifications for how to deploy and write services are still unclear. The target SOA is illustrated in Figure 6.

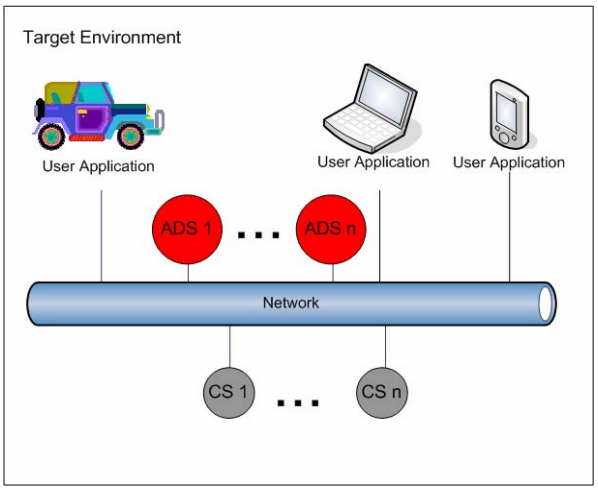


Figure 6: Physical View of the Target SOA

Figure 6 shows that the SOA includes common services (CS) that are to be used by user applications and application domain services (ADS). The SOA owns the interfaces for the common services. The environment then allows for a set of ADSs which will derive their requirements from user applications. Groups within the DoD are invited to submit proposals for services to meet these requirements, either by building them from scratch or by migrating them from legacy components. These requirements then need to be analyzed in detail and matched to existing functionality to determine what can be used as-is, what has to be modified, and what has to be new development.

Even though the full details of compliant services for the SOA have not yet been worked out, the SOA imposes a number of constraints on organizations that are developing ADSs from existing legacy components. Some of the constraints/requirements for developers of ADSs include:

1. An ADS needs to be self-contained, that is, it should be able to be deployed as a single unit.

   As expressed earlier, the reason why SOAs have been a success, especially in industry, is because they provide standard interfaces to legacy systems, while these systems remain largely unchanged.

   This is not the case in the target SOA. Services need to be stand-alone so that they can be deployed as needed on standardized platforms. In a legacy component, functionality that has been identified as part of a service needs to be fully extracted from the system, including code that corresponds to shared libraries or the core of a product line.

2. In the target SOA, an ADS has to be able to be deployed on a Linux operating system.

   For Windows-based legacy components this could be a problem, especially if there are dependencies on the operating system through direct system calls or if there is a dependency on commercial products that are only available for Windows systems. Ideally, system calls should be eliminated. If it is not possible, they should be evaluated to see if there are equivalents in the Linux operating system or if this functionality is part of one of the common services.

3. All services will share a common data model and all data will be accessed through a Data Store common service.

   The need for a common data model is driven by a desire for information to be shared and understood by all user applications. As a result, services will no longer define internal data. All data will be defined as part of the common data model. Legacy components need to replace all dependencies on databases and file systems with calls to the data store service and make sure that all the data they need is part of the common data model.

4. An ADS will use the Discovery common service to find and connect to other services.

   If the ADS will rely on other services, code to discover and connect to these services will have to be written. Once the service is developed it needs to be advertised. This is done by registering the service with the naming service. Once this advertised service has been registered, other applications that wish to use this service will perform a discovery on the available services and choose which service(s) they desire to use.

5. An ADS will use the Communications common service for communicating with other services.

   The target SOA provides tools for generating data readers and data writers that will take incoming and outgoing data and format it accordingly.

Preliminary Analysis

The current system, written in C++ on a Windows operating system, had a total of about 800,000 lines of code and 2500 classes. In addition, the system had dependencies on a commercial database and a second product for visualizing, creating, and managing maps. Both commercial products have only Windows versions.

We met with the contractor and representatives of the government to focus on a limited number of legacy components and to select criteria for further screening. We focused on seven potential services that the government team had previously identified as part of its initial analysis of ADS requirements. These seven potential services

contained 29 classes. The 29 classes that we selected enabled us to focus on potentials for high payoff. In conjunction with the team, we developed criteria for screening the potential reusable components.

Given the known and projected constraints of the target SOA, we performed a preliminary analysis of the legacy components using OAR [1] to provide a set of initial reuse estimates.

From this preliminary analysis we found that there was not adequate high level documentation. Most of the documentation was in the form of code comments and from a tool *DOxygen* which can extract after-the-fact data from the C++ code, such as classes, attributes, dependencies, and comments. However, during the analysis we found that the DOxygen tool only picked up first-level dependencies. This indicated that the coupling and the amount of code that was used by each class was higher than could be estimated from the existing documentation. There was also no consistent programming standard, leading to idiosyncrasies between different programmers. This increased the difficulty of our analysis, and it would also increase the difficulty of any reuse. As might be expected from a relatively recent object-oriented system, we found the overall cohesion to be high. The contractor provided estimates for converting the components into services, based on a set of simplifying assumptions on the actual make-up of the target SOA and the final set of user requirements.

Because of the inadequacies that we found in the architecture documentation, and the underestimation of the amount of code used by the potential services, there remained a number of gaps in our understanding of the system. For example, it was mentioned that one of the services made extensive use of the data model. This data model had over 1000 classes and was used by every class included in the potential services. Even though our analysis did not initially focus on the data model, because of its size it now represented the largest potential source of reuse in our study. However, the constraints of the target SOA may not enable the reuse of the data model.

As a result, it was not possible to accurately know how many other classes are used by a specific service. In addition the estimates for rehabilitation of the legacy components would have been understated. For example, the calls to user interface code would need to be removed, and it would be necessary to know where these are located.

To get a better understanding of these issues we performed a code analysis and an architecture reconstruction.

Code Analysis and Architecture Reconstruction

To address these issues, we first analyzed the code through a code analyzer "Understand for C++". This analysis provided:

- Data dictionary
- Metrics at the project, file, class, and function level
- Invocation tree
- Cross reference for include files, functions, classes/types, macros and objects
- Unused functions and objects

The code analysis enabled us to produce input for the architecture reconstruction tool that would identify dependencies.

As mentioned earlier, there were inconsistencies in the quality and documentation between different parts of the code that made the analysis complicated:

1. Since there was not a consistent coding standard, we could identify individual differences between programmers.
2. Some parts of the code were much more difficult to navigate, with less cohesion and a more awkward file organization. Naming standards were different for files, classes, attributes, and method names. Code organization styles were different.
3. The organization of files was not standardized either. For example, it is not clear why some files that do not perform user interface (UI) functions are located in UI folders. Another example is that some *include* files are with the code files and others in a separate folder. Some files contained more than one class and there are no clear criteria for when this is allowed.

Despite all these difficulties, that required us to gain quick high-level familiarity with the code, we were able to produce the input for the architecture reconstruction tool.

We next conducted the architecture reconstruction using ARMIN. To begin the architecture reconstruction, we took the output from the code analysis, and performed a focused analysis of the as-built architecture structure.

We aggregated the code into several groups

- One for each service analyzed
- One for code directly dependent on the commercial mapping software
- One for user interface code
- One for the rest of the code—data model, base classes, utilities, and code that did not belong to any of the above groups

Figure 7 shows the initial component view obtained using ARMIN.

8

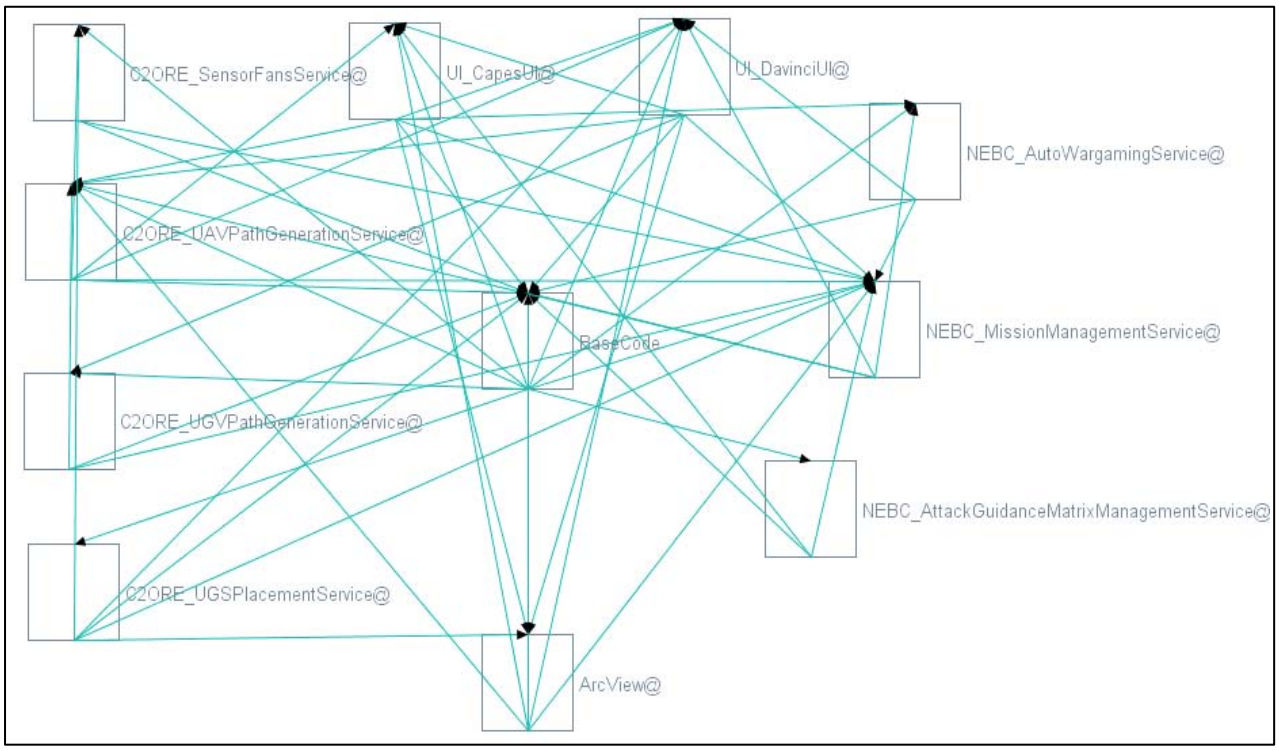


Figure 7: Component View of the Command and Control System produced by ARMIN

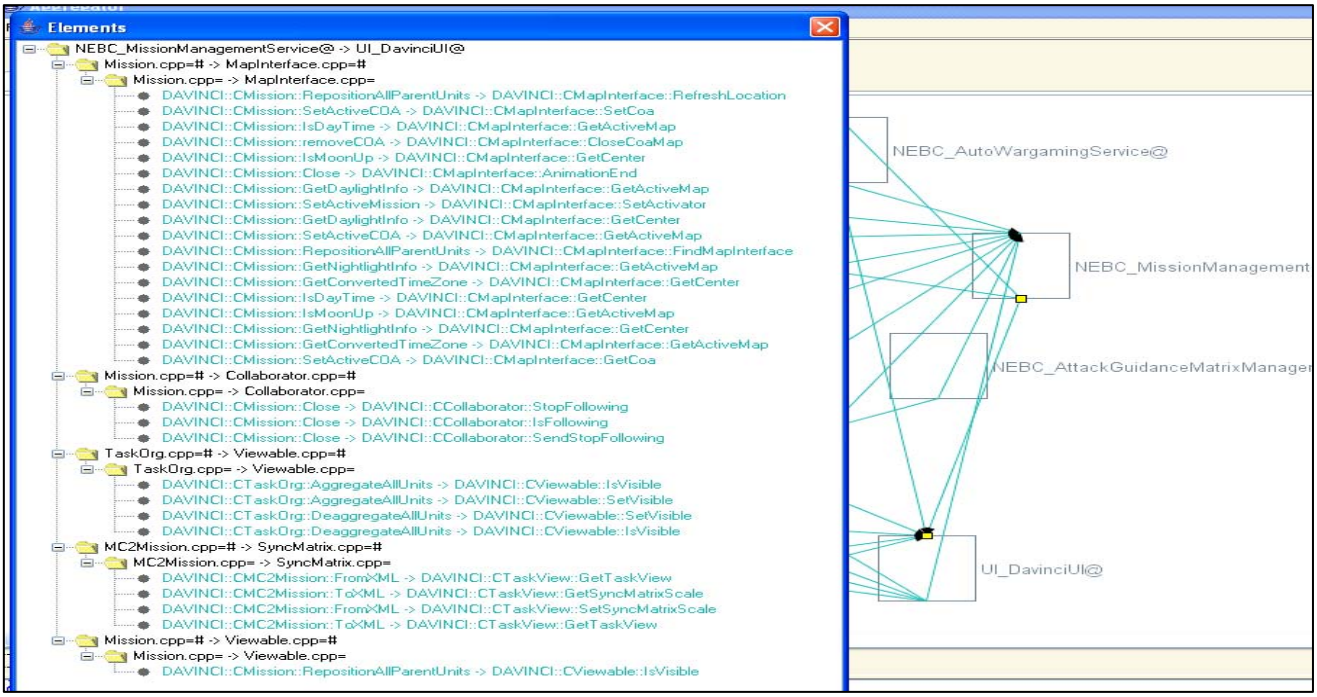


Figure 8: Dependencies between Potential Services

In our analysis, we were interested in

- Dependencies between services and user interface classes
- Dependencies between services and the commercial mapping software
- Dependencies between services
- Dependencies between the services and the rest of the code that mainly represented the data model

Figure 8 shows an example of dependencies uncovered by the analysis. We found a substantial number of undocumented dependencies between classes indicating a higher level of risk and difficulty for the migration effort than had been apparent from the preliminary analysis. For example the preliminary estimate for one of the potential services was 4859 LOC. After the architecture reconstruction analysis that uncovered undocumented dependencies, we found that this service would actually consist of 66,178 LOC.

The architecture reconstruction also enabled us to document the central role of the data model, and to identify it as a potentially valuable reusable component, even though it had not been identified during the initial analysis. However, this finding was tempered by the fact that in the target SOA environment, potentially all services will have to use a common data model. If this is the case, all elements of the data model will have to be mapped to existing elements of the common data model. Negotiations would have to take place to make sure that data elements that are needed by the services become part of the common data model.

The architecture of the system is an example of the application of the *Model View Controller (MVC)* pattern. The architecture reconstruction found undocumented violations of the MVC architecture which would need to be addressed in any migration effort—specifically calls from the model to the view.

An illustration of how the change from a standard system development effort to an SOA can have unanticipated impacts can be seen from the product line approach that was taken in the initial development of the system. The product line approach was an excellent choice for the current application; however, it might increase the difficulty of the migration effort due to the potential requirement for services to be stand-alone for easy of deployment in the target SOA. The large dependency on base code and multiple levels of inheritance make it difficult to isolate services. A potential solution to this problem would be to consider each service in itself as part of a product line, but this of course would require the set of core components to be potentially redefined.

Implications of the Analysis

In looking at the potential for the service migration, the preliminary analysis suggested that the current legacy code represents a set of components with significant reuse potential. However, because the current legacy system does not have sufficient architecture or other high level documentation, it was difficult to understand the "big picture" as well as dependencies between different classes. The architecture reconstruction provided an "as-built" representation of the structure of the system and its dependencies. It suggested that the significant dependencies between classes will make reuse and deployment of services more difficult. If the migration to service effort moves forward, the results of the architecture reconstruction can enable a starting point for understanding how to disentangle dependencies

In addition there is a risk in making migration decisions now because the target SOA has not been fully defined. While its overall structure has been defined, many of the specific mechanisms for interacting with it are still pending. Thus, it is not yet clear what the requirements for being a service in this environment will be in 12 or 18 months.

We also recommended that the government organization require the following changes from its contractors to make reuse of its legacy components more viable:

- Suitable set of architectural views
- Consistent use of programming standards
- Documentation of code to enable comments to be extracted using an automated tool
- Documentation of dependencies, especially when they violate architecture paradigms

In addition, we recommended that the government organization take a proactive approach in working with the developers of the target SOA to understand implications of the current and evolving SOA plans for how potential interacting services need to be developed. The government organization should also work closely with the developers of the applications that will be using these services. Even though the technical part of the communication will be handled by a common service, what data is transferred during that communication has to be negotiated—the contents of both the request and the response message that is communicated between the application and the service need to be defined.

5. Conclusions and Next Steps

We found that the use of architecture reconstruction to understand the as-built system provided an essential step in making decisions on the migration of the legacy components to services. The initial task of determining how to expose functionality as services, while seemingly straightforward, can have substantial complexity. Our conclusions to the client, while not definitive, did point out a number of issues that they had not previously considered. The use of architecture reconstruction techniques, in conjunction with other analytical methods, provides an essential set of analytical methods for decision-making.

We are currently developing a method that integrates architecture reconstruction with other analytical methods for reuse decision making, such as OAR. This method, Service-Oriented Migration and Reuse Technique (SMART) focuses on the specific issue of migration of legacy components to an SOA. The next steps in the development of the method will be to:

- Include a greater number of specific tools that directly address the SOA concerns that need to be addressed when exposing functionality as services. We are developing the Service Migration Inventory (SMI) as the first of such tools.
- Incorporate decision rules on when it is most useful to include the code analysis and architecture reconstruction steps as part of the process.
- Make the process repeatable so that it can be used by the wider community. The tools and decision rules being developed are a first step in developing a repeatable process.

REFERENCES

1. Bergey, J.; O'Brien, L.; and Smith, D. "Using the Options Analysis for Reengineering (OAR) Method for Mining Components for a Product Line," 316-327. Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2). San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002.
2. Bowman, T.; Holt, R. C. and N. V. Brewster, Linux as a Case Study: Its Extracted Software Architecture. *Proceedings of the International Conference on Software Engineering*, Los Angeles, May 1999.
3. The Dali Architecture Reconstruction Workbench: http://www.sei.cmu.edu/ata/products_services/dali.html
4. Eixelsberger, W.; Ogris, M.; Gall, H. and Bellay, B., Software architecture recovery of a program family, *Proceedings of the International Conference on Software Engineering*, pp 508 –511, Kyoto Japan, April 1998.
5. Finnigan, P. J.; Holt, R.; Kalas, I.; Kerr, S.; Kontogiannis, K.; Mueller, H.; Mylopoulos, J.; Perelgut, S.; Stanley, M. and Wong, K., The Portable Bookshelf, *IBM Systems Journal*, Vol. 36, No. 4, pp. 564-593, November 1997.
6. Guo, G.; Atlee, J. and Kazman, R., A Software Architecture Reconstruction Method, *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, Texas, February 22-24, 1999 pp 225-243.
7. Harris, D.R.; Reubenstein, H. B. and Yeh, A. S., Recognizers for Extracting Architectural Features from Source Code, *Proceedings of the 2nd Working Conference on Reverse Engineering*, 1995.
8. Harris, R.; Reubenstein, H. B. and Yeh, A. S., Reverse Engineering to the Architectural Level, *Proceedings of the International Conference on Software Engineering (ICSE)*, pp 186-195, April 1995.
9. Imagix Corporation's Imagix 4D: http://www.imagix.com/
10. Kazman, R. and Carrière, S. J., Playing Detective: Reconstructing Software Architecture from Available Evidence, *Journal of Automated Software Engineering,* pp 107-138, April 1999.
11. KLOCwork inSight: http://www.klocwork.com/Accelerator.htm
12. Kontogiannis, K. and Zou, Y. Reengineering Legacy Systems Towards Web Environments, in "Managing Corporate Information Systems Evolution and Maintenance", Idea Group Publishing, Hershey, PA, USA. pp. 138-146, 2004.
13. Krikhaar, R. L., Software Architecture Reconstruction, *Ph.D. Thesis,* University of Amsterdam, 1999.
14. Laine, P. K., The Role of Software Architecture in Solving Fundamental Problems in Object-Oriented Development of Large Embedded Systems, *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*. Amsterdam, The Netherlands, pp 14-23, August 28-31, 2001.
15. Litoiu, M. Migrating to Web Services: Latency and Scalability. *Proceedings of the Fourth IEEE Workshop on Web Site Evolution (WSE'02), 2002.*
16. Mendonça, N. C. and Kramer, J., Architecture Recovery for Distributed Systems, *SWARM Forum at the Eight Working Conference on Reverse Engineering,* Stuttgart, Germany, October 2001.
17. The Portable Bookshelf: http://swag.uwaterloo.ca/pbs/
18. Riva, C., Reverse Architecting: An Industrial

Experience Report, *Proceedings of the Seventh Working Conference on Reverse Engineering*, Brisbane, Australia, pp 42-50, November 23-25, 2000.

19. Sartipi, K. and Kontogiannis, K., A Graph Pattern Matching Approach to Software Architecture Recovery, *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001),* Florence, Italy, pp 408-419, November 7-9, 2001.

20. Scientific Toolworks Inc's Understand for C/C++/ Java /Fortran/Ada: http://www.scitools.com/

21. Sneed, H. and Sneed, S. Creating Web Services from Legacy Host Programs. *Proceedings of the Fifth IEEE Workshop on Web Site Evolution (WSE'03),* Amsterdam, Netherlands, 2003.

22. Stoermer, C., O'Brien, L. and Verhoef, C. Moving Towards Attribute Driven Software Architecture Reconstruction. *Proceedings of 10th Working Conference on Reverse Engineering, 2003,* WCRE 2003, Victoria, British Columbia, 2003.

Using MAP for Recovering the Architecture of Web Systems of a Spanish Assurance Company

Rafael Capilla
Dept. of Informatics and Telematics
Escuela Superior de Ciencias Experimentales y Tecnología
Universidad Rey Juan Carlos, c/ Tulipán s/n, 28933, Madrid, Spain
rafael.capilla@urjc.es

Abstract

In this work we describe the steps of the MAP method to recover the software architecture of web-based application of a Spanish assurance company. The main goal is to reduce maintenance costs avoiding software components with similar functionality. We describe a case study of a Spanish assurance company to obtain a single software architecture for several applications belonging to the same domain in order to promote the production of reusable components under a product line context.

1. Introduction

The maintenance of web systems is usually performed in a short time because customers usually expect to obtain results in a few days or weeks. Classical web maintenance operations can be divided into several categories, such as: modification and updating of web pages, populate databases, graphical design changes, middleware development or even other tasks non related to software development (e.g.: server maintenance operations). Many times and due to time pressure, web maintenance operations are not well documented and changes are not properly reflected in the design. A lack of system understanding motivated by old or nonexistent software designs can lead to a general delay of time-to-market of new products or system versions. In addition to this, software components with similar functionality are built for similar systems and the lack of reusable software becomes common to many in software projects. Therefore, the need to employ software architectures [2] is a key aspect in order to reduce both development and maintenance costs.

In this work we employed the MAP (Mining Architectures for Product Lines) [14] [17] method from the SEI (Software Engineering Institute) to perform an architectural recovery process that help us to discover the common and variable aspects of two existing web systems in order to obtain a valid software architecture [2]. The aim of the MAP method [14] [17] is to reconstruct the architectures of similar products and discover the variable aspects in related software products. As a result of this process, we should be able to decide if we can move or not to a product line approach in order to develop new products in a faster way employing reusable components. One reason for using MAP is that old software designs are not flexible enough to support the variable aspects needed to support the quick creation of new products.

2. A Case Study of a Spanish Assurance Company

In our study we analyzed the systems of a Spanish SME assurance company which has been involved in the development of 14 web projects. Four of these projects belong to the intranet of the company which constitutes the core business. In this case study we applied the MAP method to two web applications: *a training service system and a customer telephone service*.

The first application manages a set of training services that include courses, exams, news, etc., all of them stored in a SQL Server database and replicated in MS-Access tables. The system has an SMS manager module used for sending messages to mobile phones from the intranet.

The second application supports the first one providing access to the users under an intranet. Several

categories of permissions, users and groups exist in the database for accessing different services and controlled by an authentication module. This authentication module uses the Windows 2000 Active Directory for validating users through the intranet. Both systems share common database technologies such as: SQL Server and MS-Access. A set of COM+ DLLs provide the functionality to access services to data employing transactional facilities given by the COM+ technology. One of the components implements business logic functions needed by the intranet to access the data stored in the SQL Server database, while others are used to access the data belonging to the training services. The rest of the modules are developed using the Active Server Pages (ASP) and some of them employ JavaScript functions for generating dynamic menus. The web server employed is the Microsoft's Internet Information Server (IIS) with the Mail Server facilities. The summary of the assets included in each module is the following: 144 ASP pages, 25 JavaScript functions, 9 HTML pages, 56 VB Database functions, 33 tables for the Training database and 7 tables for the Intranet database. Table 1 shows the development effort for each module from both applications, including the time spent in testing and correcting each module. Also, the time spent integrating the system is included.

Table 1. Development, correction and testing effort spent in the modules of both systems

Name of Module	Development time
DAL_INTRANET	48 h / 1 man
BUS_INTRANET	48 h / 1 man
DALFORM	160 h / 2 men
BUSFORM	120 h/ 2 men
SMS Manager	5 h / 1 man
Training Module	240 h / 3 men
Administration	160 h / 1 man
SMS Module	1 h / 1 man
Mail Module	2 h / 1 man
Includes	80 h / 1 man
Library of functions	40 h / 1 man
Menu Generator	160 h / 2 men
Style Sheets (CSS)	8 h / 2 men
Authentication	80 h / 1 man
Intranet Database	16 h / 1 man
Training services Database	24 h / 1 man
TOTAL HOURS	1.192

Maintenance processes are time consuming in both applications because they are closely related to each other. Also, the existence of old designs in both software systems and the lack of reusable components that can be used in several projects of the company, motivated the need to count with a common software architecture able to reduce the effort employed in development and maintenance tasks. The MAP approach became a good choice to obtain such common design.

3. Applying MAP to Web-based Systems

The similarity between the two systems analyzed makes them suitable as inputs of the MAP method to obtain an architecture common for both systems. The steps of the MAP method we carried out are the following.

Preparation: As we mentioned in the introduction of this work, the motivation for moving to a product line approach [9] is to reduce the development and maintenance effort of two related web systems and try to produce a set of components that can be reused by other applications of the company. We applied the MAP method to the two systems described in section 2. Both systems are in the same domain and closely related such as we have mentioned before. Also, they are representative and share many commonalities with the other web applications of the company.

Extraction: In this step, first we identified the candidate subsystems and afterwards we extract the information of a representative group of assets from both systems. We didn't use a tool such as Dali workbench [3]. Instead we employed a visual version of the *Unix diff* program (Visual Diff) for comparing the assets and we stored the information gathered in an MS-Access database. In this way we can say that the code analyzed from web-based applications is not too complex to understand and the comparison process between the source code files was not too difficult. We managed the inspected files with the programming tools provided by Visual Basic and Visual Interdev. Also, we used legacy documentation to gather some information about the COM+ components.

Composition: Here we tried to identify components views based on the information gathered in the previous step. We identified not only relationships among the components of each system but also relations among common components for both systems. To establish appropriate relationships we grouped similar assets by functionality and we separated those ones common for both systems. We identified around 17 relationships such as table 2 shows (some of them are grouped in the same row).

Table 2. Relationships between the components of both systems obtained by the MAP method

Module	Related to Module	Description
DAL_INTRANET (DLL COM+)	Intranet Database	DAL_INTRANET accesses to the intranet database
BUS_INTRANET (DLL COM+)	DAL_INTRANET	The intranet business logic is implemented by the BUS_INTRANET component which accesses the data through the DAL_INTRANET component
DALFORM (DLL COM+)	Training Database	DALFORM access to the training database
BUSFORM (DLL COM+)	DALFORM	Establishes the same type of relationship as mentioned for the BUS_INTRANET component but for accessing the training database
SMS Manager (DLL VB)	Intranet Database	SMS Manager stores the logs of the service in the intranet database
Training Module (ASP)	BUS_INTRANET	The training module implements the presentation layer of the BUSFORM component
Administration (ASP)	BUSFORM	The administration module implements the presentation layer of the BUS_INTRANET component
SMS Module (ASP)	SMS Manager	The SMS module provide suitable ASP pages for sending SMS messages
Mail Module (ASP)	Mail Server	The mail module provide e-mail functions for sending electronic mail
Includes (ASP)	Library of functions Menu Generator, Style Sheets (CSS), Administration, Training Module, SMS Module SMS, Mail Module, Authentication	The purpose of the includes modules is to establish relations among all the modules of the intranet
Web Server (IIS)	Authentication (Active Directory)	The IIS modules is related with authentication processes for granting or denying access to users

Qualification: Once we established the basic among the components, we search for known architectural styles to perform a mapping process within the relations obtained before. We used Client/Server and layered architectural styles to establish architectural views for both systems and we tried to group them in a single view. Also we analyzed several quality attributes to establish appropriate relationships. These quality attributes are the following:

- **Security**: Because the authentication module is a requirement against unauthorized users.
- **Efficient**: The implementation needs to guarantee the efficiency of the solution provided (e.g.: based on speed or security aspects).
- **Ability to perform transactions**: This is a requirement for database operations supported by the COM+ modules. Strictly speaking this is not a “quality attribute” but has influence on other quality attributes such as “speed” or “safety”.

Evaluation: As a result of the steps performed we obtained two similar architectures connected by common modules. Therefore, we decided to provide a unique software architecture obtained by the fusion of both candidates. To achieve this we had to define appropriate variation points able to customize the architecture to produce similar products. The variation points defined are based on web-based code such as customizable features of HTML elements (e.g.: tables, frames), JavaScript menu functions and ASP code. For instance the variation points [5] of a HTML table element can be defined based on the columns, rows, cell spacing, color, or vertical align parameters among others. Also, a variation points for selecting different database operations and different database systems were defined. Different classes of user and user’s grants as well as mail options constitute candidate variation points to be included in the software architecture.

Follow-on: The activities we recommend as follow-on guidelines, is to develop the core assets [9] for the product line. These assets can be engineered from scratch or even reused using the SEI's OAR (Option Analysis for Reengineering) method [4] [14]. Other possibility is to wrap existing assets to include the variation points defined in the architecture. In addition and for web systems, we recommend a lightweight product line approach [1], such as mentioned in [7] to produce the reusable assets. In the approach described in [7] we avoided complex processes such as domain analysis [13] [16] [18] in favor of reverse engineering ones [11]. As a result, the software architecture proposed as a candidate to establish a product line model is shown in figure 1.

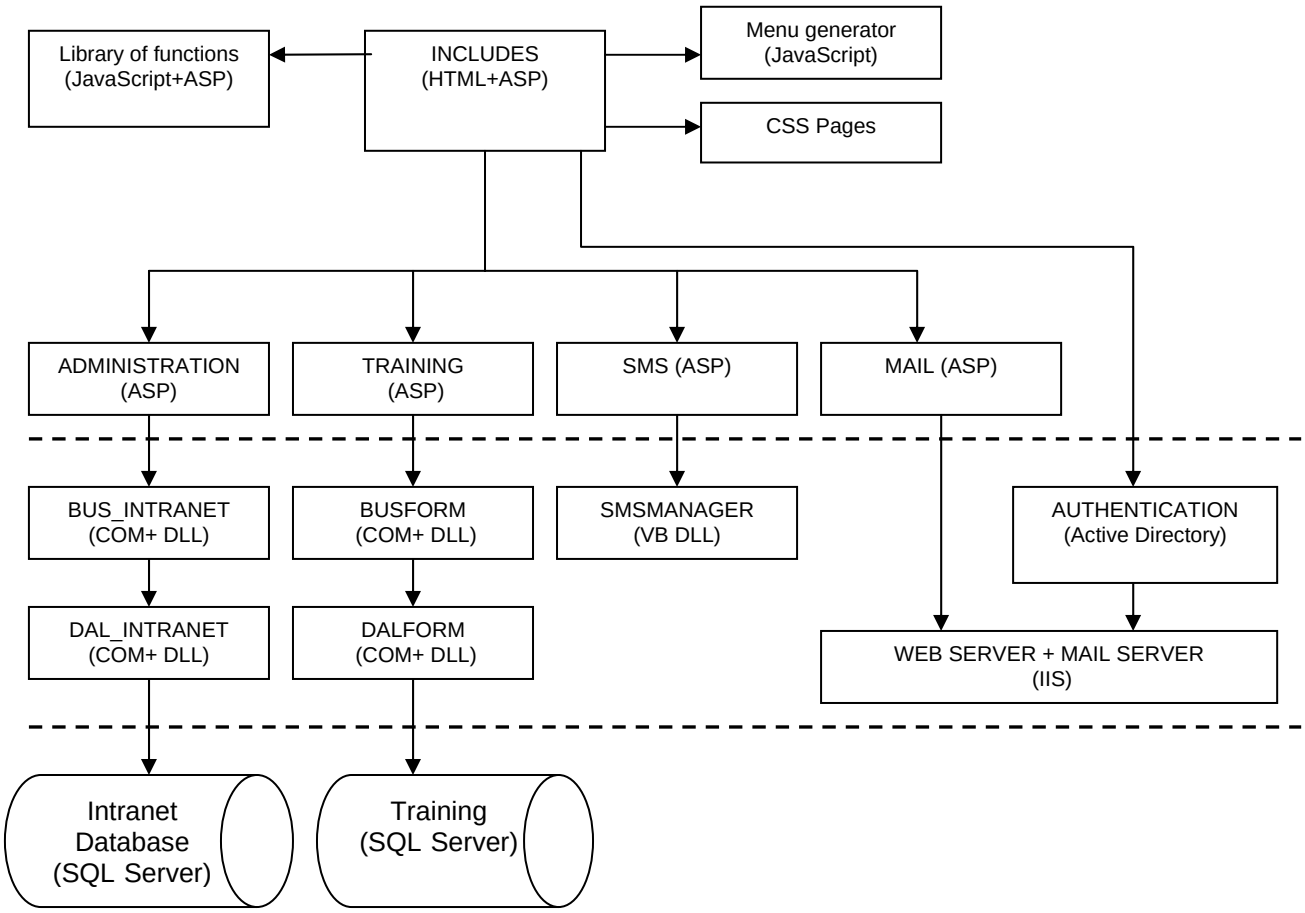


Figure 1. Software architecture obtained from the MAP method for both analyzed web-based systems

Applying the MAP method took over 50 hours. Two graduate students and one senior software engineer did this work conducted by an instructor. Also, we employed over 6 hours training the team about software architectures, reuse, variability issues and product line concepts.

4. Application to Reuse

In the recent past years, several experiences and industrial efforts have proved that using product line approaches and well defined software architectures increase the reusability and maintainability of software systems. Well-known mechanisms such as software variability are used to produce both reusable assets and customizable software products faster than other traditional approaches. A product line model [12] shares customizable and reusable core components that are used in the construction of similar systems. In this case study we didn't performed an exhaustive analysis of the reuse results by the two systems analyzed as well as by other systems of the company but the main results

we can extract after the MAP process towards reuse are the following.

1. A single software architecture that can be used by several systems of the company. We estimated that at least 7 software applications can be represented using the architecture of figure 2 and performing slight modifications in the design.
2. Definition of appropriate variation points [5] [6] to customize the reusable components and to drive the evolution of the systems in the future.
3. Construction of 6 reusable core assets for the two systems analyzed during the MAP process and probably, this number will increase with new reusable assets from other 5 systems.
4. Reduce the maintenance effort not only for source code. Also, the software architecture facilitates the maintenance of old designs having only a single one.

5. Conclusions

In this work we have described a MAP experience for several web-based systems of a Spanish assurance company.

The first goal was to obtain a common software architecture for two web-based products that can be used by other systems of the company. One of the reasons to obtain suitable software architectures is because in certain cases software designs are too rigid to support the evolution and changes needed by the users or by new requirements. The need for flexible architectures able to support variability of its parts, results key to predict future changes in the development of new products. One conclusion we can extract applying MAP is that it becomes useful when the domain is well scoped [8], such as the case of our company. Respect to the variation points, sometimes is difficult to decide where to place and define the variations; in particular if we haven't decided which assets will be core and non-core for the product line. The architect should solve such architectural problems in order to initiate the product line as soon as possible.

The second goal was to establish appropriate guidelines for moving to a lightweight product line instead of using other more complex models [10] [18] to promote the reusable features of the systems. These guidelines for promoting reuse using a product line approach should be given in the follow-on step of the MAP method. In particular and based on [7], we advocate the use of lightweight product lines to be more suitable for web systems because they employ less complex processes to set-up the product line and web products can be engineered faster. In this way, estimating time to market in web development is quite different than in the traditional software development processes [15]. But for making the investment of the product line viable, several systems should be engineered. In the case of the company on which we proved the MAP process, the architecture obtained can be applied to at least 7 of the 14 web projects. This makes sense the use of employing a product line model [6] [12].

As stated in section 4, the results applying a product line model to the systems of the company over time will report the reuse feedback needed to be employed in the construction of new software systems.

References

[1] S. Bandinelli, "Light-weight Product-Family Engineering" *4th International Workshop on Software Product-Family Engineering*, pp. 327-331, European Software Institute, Bilbao (Spain), 2001.

[2] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison Wesley, 2003.

[3] J. Bergey, L. O'Brien. And D. Smith "Mining Existing Assets for Product Lines", CMU/SEI-2000-TN-008, Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA (USA), 2000.

[4] J. Bergey, L. O'Brien and D. Smith. "Option Analysis for reengineering (OAR): A Method for Mining Legacy Assets", CMU/SEI-2001-TN-013, Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA (USA), 2001.

[5] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink and K. Pohl, "Variability Issues in Software Product Lines", Software Product-Family Engineering, LNCS vol 2290, Springer-Verlag, pp. 13-21, 2002.

[6] J. Bosch, Design and Use of Software Architectures, ACM Press-Addison Wesley, 2000.

[7] R. Capilla and J. C. Dueñas, "Light-weight Product Lines for Evolution and Maintenance of Web Sites" 7th *IEEE Conference on Software Maintenance and Reengineering*, Benevento (Italy), pp. 53-62, 2003.

[8] P. Clements, "On the Importance of Product Line Scope", Software Product-Family Engineering, LNCS vol 2290, Springer-Verlag, pp. 70-78, 2002.

[9] P. Clements and L. Northrop, Software Product Lines, Addison-Wesley, 2002

[10] J. M. DeBaud and P. Knauber, "Applying PuLSE for Software Product Line Development" *2nd European Reuse*

Workshop, pp. 73-94, Madrid, Spain, 1998.

[11] A. Maccari and C. Riva, "Architectural Evolution of Legacy Product Families", Software Product-Family Engineering, LNCS 2290, Springer-Verlag, pp. 64-69, 2002.

[12] F. A. Maymir-Ducharne, "Product Lines, Just One of Many Domain Engineering Approaches", *A NASA Focus on Software Reuse*, 1996.

[13] A. Mili and S. M. Yacoub, " A Comparative Analysis of Domain Engineering Methods: A Controlled Case Study", Proceedings of Software Product Lines, Economics, Architectures and Implications, at 22nd International Conference on Software Engineering, Fraunhofer IESE TR-070.00/E, 2001

[14] O'Brien, L., Smith, D., MAP and OAR Methods. Techniques for Developing Core Assets for Software Product Lines from Existing Assets, CMU/SEI-2002-TN-007, Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA (USA), 2002.

[15] D. J. Reifer, Web Development: Estimating Quick-to-Market Software, IEEE Software, 57-64, 2000.

[16] W. Schäfer, R. Prieto-Díaz and M. Matsumoto, Software Reusability, Ellis Horwood, 1994.

[17] C. Stoermer and L. O'Brien. "MAP: Mining Architectures for Product Line Evaluations", Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, 2001.

[18] D. M. Weiss and C. T. R. Lai. Software Product-Line Engineering: A Family-Based Software Development Process, Ed: Addison-Wesley, 1999.

Workshop 2.4: Workshop on Design Issues for Software Analysis and Maintenance Tools

Tool-Supported Realtime Quality Assessment

Florian Deißenböck, Markus Pizka, Tilman Seifert*
Software & Systems Engineering
Technische Universität München
85748 Garching, Germany

Abstract

Maintenance costs make up the bulk of the total life cycle costs of a software system. Besides organizational issues such as knowledge management and turnover, the long-term maintenance costs are largely predetermined by various quality attributes of the software system itself, such as redundancy and adequate documentation. Unfortunately many quality defects can hardly be corrected retrospectively after they have penetrated the system. A much more promising approach than correction is to avoid decay and to preserve a constant high level of quality through a continuous realtime quality controlling process. To reduce the costs for the required frequent quality assessments, adequate tool support is indispensable. This paper proposes to integrate measurement tools into a flexible and extensible yet high performance quality assessment architecture. A key design priniciple of this architecture is to abstain from an intermediate representation or meta model that is used as a common ground for all analyses, but to allow for more flexible implementations of different analyses.

1 Introduction

Software maintenance activities typically consume 80% of the total cost of a software system [2]. While 80% sounds dramatic, the interpretation of this number, its reasons and consequences is not as obvious as it is often regarded. For example, rather low annual maintenance costs of 10% of the original development costs over a period of 30 years sum up to 75% over the complete life-cycle. Hence, 80% by itself might not indicate a problem but be a side-effect of the long-term success of a software system. Likewise, spending 75% of the budget on maintenance activities [11] does not justify the term "maintenance crisis" without a detailed view on the actual situation and context. However, it is clearly a problem if maintenance activities, such as adding new functionality, consume excessive amounts of time, or if change requests are primarily corrective instead of perfective or adaptive ones [9].

Besides organizational issues, e. g. qualification and turnover, the major reason for such undesirable effects are quality defects. Standards such as ISO 9126 [7], define *maintainability* by means of a set of quality attributes, such as *analyzable, changeable, testable, and stable*, or more detailed ones such as consistent and concise naming [4].

Violations of these quality criteria may either be introduced during initial development or caused by long-term decay [5]. Once these defects have found their way into the system, they are usually very hard to correct. Obviously, restructuring a weak architecture requires extensive resources but even assumed minor changes, such as the removal of copied and pasted code duplicates[1] can easily become excessively complex.

Reel states that "by the time you figure out you have a quality problem, it is probably too late to fix it" [10]. Therefore, we claim that it is necessary to continuously and closely monitor the quality of software systems in order to prevent defects from creeping into the system as far as possible. We argue that given appropriate tool support and a certain amount of process discipline, the cost of maintaining high quality can be diminishing low and will deliver rapid pay-off, often even within the development phase, already. The tool concept proposed in this paper is based on the seamless integration of different measurement tools into a flexible, extensible, yet efficient quality assessment architecture. This setup allows to assess the quality of a software system in real-time and paves the ground to establish a continuous quality controlling process.

Outline After a discussion of related work on quality measurement tools in section 2 we will state a set of requirements for suitable tool support for real-time quality controlling with respect to maintainability in section 3. The design

*Part of this work was sponsored by the German Federal Ministry for Education and Research (BMBF) as part of the project VSEK (Virtual Software Engineering Competence Center).

[1] clone removal

and implementation of our tool *ConQAT* (Continuous Quality Assessment Tool) that is detailed in section 4. Section 5 summarizes our experiences with *ConQAT*.

2 Related Work

Commercial vendors as well as the open source community offer a plethora of diverse software analysis and quality assessement tools.

An approach taken frequently is the construction of a *facts database* or an intermediate *meta-model* representing an abstraction of the source code. Commonly used levels of abstraction are for example an *abstract syntax tree* (AST) or a *call-* or *dependency-graph* of the system. Typically object-oriented [8] or relational models [1] are used to implement the selected level of abstraction. Albeit great differences in the detailed design of the various *facts* or *meta-model* based approaches to software analysis, all of these approaches preprocess the input before performing the actual analysis. During this preprocessing stage the system under investigation gets parsed and transformed into the format of the meta-model. All analyses and assessments are then carried out on the meta-model.

Although this well-structured tool design is consequent and elegant from a software engineering point of view, it has a major drawback: it rigidly defines a certain level of abstraction and thereby limits the range of possible analyses. Code duplication checks, for example, can't be performed on the dependency graph. As important quality aspects are of very diverse nature and rely on different information this problem is typically circumvented by offering multiple layers of abstraction for the different types of analyses. But, this means that all information needed to construct the complex multi-layer meta-model needs to be acquired for the entire system, which in turn renders building the meta-model a very expensive task. In practice, preparing the meta-model of a large scale system often takes several hours which is unacceptable for realtime quality assessment. In fact, these tools become used by quality experts for rather infrequent in-depth investigations of certain quality criteria. They are not suited for the integration into a continuous quality controlling process.

Besides *facts* and *meta-model* based approaches, there are numerous metric tools. They come as stand-alone tools or plug-ins for development environments like Eclipse (e. g. *Metrics*[2]). This range of products is supplemented by a number of assessement (or audit) tools like PMD[3] which usually offer batch and interactive operation modes, too. Some of the available tools are designed as extensible platforms which may be augmented with custom analyses allowing a centralized view of the results. However, they fail to provide means to compose more complex analyses from more simple analysis modules. Though, literature on software quality [3, 7] clearly points out that quality is a complex and diverse matter which can only be assessed by analyzing and aggregating a great number of influencing factors. Therefore the composition of different analyses that create a holistic view on a system's quality is a crucial feature of a quality analysis tool.

[2] `http://metrics.sourceforge.net/`

[3] `http://pmd.sourceforge.net/`

3 Tool Requirements

3.1 Target Process

The success of quality management largely depends on the way it is embedded into the development and maintenance process. We envision a quality management process with the following key features.

The process allows to detect small deviations from the target quality, so that corrective action can be taken immediately, and it allows to do so early and continuously, i. e. "in real-time of development". It provides and uses a set of information about the current quality status of the system. This information must be complete and detailed on one hand to be of direct value to the developers, on the other hand it must provide an aggregated view on the same information to give a quick and accurate overview over the status of the system. It must be possible to tailor this information to the specific needs of the project. The information must be up-to-date and available at any time; the collection of data must not affect development tasks.

The whole process is structured into two parts, the operative and the strategic part. The operative part deals with measuring quality attributes and changes thereof in the daily development tasks. The strategic part identifies the criteria to be checked.

3.2 The Right Criteria

For installing an effective quality management process, the real challenge is to find the right criteria that allow to draw an accurate picture of the quality of a system. We don't want our tool and process to be driven by measures just because they are easily realized. It does not really matter whether some source code lines contain more than 80 characters, and the *coupling between objects* measure alone does not support a substantial statement about those dependencies that make maintenance a hard job.

We rather try to identify the factors that have a major impact on the maintainability in the long run. We even include factors that are not directly measurable, and instead check "manual" evaluations against a wide variety of features. For example, our three-stage source code rating (which we introduce in section 4.2) leaves the decision about the matu-

rity of the code to the developer who notes it in the source code. In a second step we can check source that is assigned as mature for completeness of JavaDoc comments, the comment ratio, appropriate repository check-in messages, successful runs and completeness of unit tests etc.

3.3 Tool Support

Following this reasoning, we derive the following key requirements for the tool suppport of the quality management process.

Static Output: The tool should work in a non-interactive, automated way with a static output so that there is no additional cost (in time, effort, or motivation) to use it. It should integrate different result types. For *ConQAT*, we decided to produce an HTML page in the nightly build that gives a detailed, integrated report about all quality attributes considered in the following.

Different Views: Information should be available in a detailed form as well as in an aggregated, brief form to satisfy the needs of different stakeholders in the project.

Flexibility: The system needs to be flexible in a way that it is easy to combine different analyzers and to configure different analysis runs that exactly match the needs of the project. In different phases and for different projects, different questions might be asked. The tool should be easily configurable to give concise answers.

Extensibility: The same argument leads to the requirement that the tool should provide an infrastructure to make extensions with new analyzers as easy as possible.

Diversity: Quality attributes can be discussed on many different levels. The tool should make no restrictions about the level of detail, the level of granularity, nor the type of the attribute of the analysis. Examples of analysis levels include the source code, the build process, the documentation, or repository properties.

4 ConQAT

To our knowledge none of the tools available completely satisfies the requirements pictured above. We therefore designed *ConQAT* from scratch. Our design considerations are led by the requirements above and by the experiences we made with existing analysis tools as well as our own prototypes.

A central decision is not to follow the approach taken by fact-databases or meta-modelling tools as we believe that a common abstraction level inherently increases implementation effort and hampers performance. Why load a complex call graph representation of the entire source code into a relational database, just to find out how often it contains the string literal "TODO"? The usual argument in favor of a common abstraction level is reduced redundant analysis steps. As *ConQAT* shows, this problem can be elegantly circumvented by using smart caching mechanisms while making sure that no superfluous analyses are carried out.

It quickly became evident that our tool must provide a flexible extension (or plug-in) mechanism to fulfill our requirements. These extensions can carry out various analyses whose results should be composable.

The main challenge here was the design of an architecture which is rigid enough to allow the efficient combination of different analyses while being flexible enough to integrate the plethora of different kinds of analyses. Detailed analysis of different extensible architectures pointed out that there is in fact a *spectrum of flexibility*. However, there's always a trade-off between the flexbility and the expressiveness of the extensions.

On one end of the spectrum you find architectures which are extremely rigid. They define a very stringent extension interface and thereby limit the extensions' expressiveness. Nevertheless they allow a flexible composition of the extensions and permit a rich infrastructure in the architectural core of the system. On the other end of the spectrum you find architectures which define a very unspecific interface and integrate their extensions only loosely. This enables extensions to be much more powerful but limits composition possibilities and inhibts a rich common infrastructure.

To obtain a better understanding of this spectrum we developed two prototypes close to both ends of it. The one on the *rigid* end basically supported a mapping from compilation units to numerical metric values. Obviously this mechanism allows very efficient composition of different analysis modules but limits the range of analysis types. It doesn't support metrics which yield anything but a numerical value (without cumbersome workarounds) and makes analyses with a granularity different from compilation units impossible. The prototype at the other end of the spectrum was more or less a web portal which allowed the extensions to contribute HTML pages with their analysis results. It should be clear that this approach allows almost unlimited possibilities for the extensions but makes a meaningful composition of different extensions nearly impossible.

4.1 Design

These considerations finally led to the design depicted in figure 1. *Processors*, *Driver* and *Libraries* are the basic elements of *ConQAT*'s architecture.

Processors To be as flexible as possible *ConQAT* analyses are composed of various extension modules, called *processors*. These processors have highly diverse tasks and are connected in a pipes-and-filter oriented style. Usually the data structures passed from one processor to the other have a tree-like format representing e. g. files and directories or

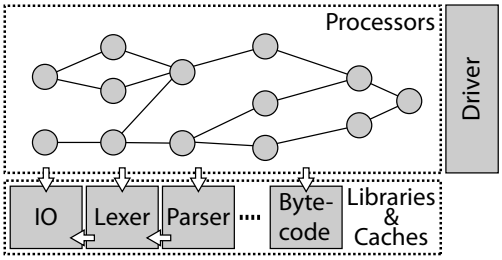


Figure 1. Architectural Overview

packages and classes. However, this is not mandatory; they may be of any possible format. Some of the current processors use lists and even single values. Typical tasks of processors include (but are not limited to):

Scoping. A processor may define the scope of particular analyses. It does this by building an appropriate object tree (e. g. representing files and directories) and passing it on to other processors.

Filtering. According to some filtering criterion a filtering processor may remove particular nodes from the tree (e. g. discard files from a certain author).

Analysis. An analyzing processor carries out a particular analysis on the elements of an object tree and annotates the tree elements with the results (e. g. lines of code or number of comment lines).

Aggregation. An aggregating processor collects values from different analyzers and annotates the tree with aggregated values (e. g. comment ratio).

Output. Finally a processor responsible for the output collects the trees and displays them in a human-readable format (e. g. HTML).

The actual analysis is defined by the composition of different processors. To allow greatest flexibility this is done via an XML-based configuration file.

Processors are implemented in Java. We use Java's new *meta data mechanism*[4] to annotate the processors with additional information that defines the mapping between a processor's implementation and its description in the configuration file. This meta data is furthermore used to automatically generate processor documentation.

Driver The driver component is responsible for interconnecting the processors as defined in the configuration file and running the analysis. During start-up the driver loads the processors' classes via reflection and uses their meta data to ensure type safety. It thereby ensures that the configuration is valid before starting the analysis.

[4] `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`

Processors are topologically sorted and then ran one after another. During execution the driver passes the result from one processor to the next. If a processor's result is used more than once the driver is responsible for cloning it. It additionally performs some monitoring tasks to provide debugging information if one of the processors should fail.

Libraries By nature, several different processors work on the same resources, e. g. different code style analyses on the AST of the compilation units. *ConQAT* offers a set of *libraries* that provide commonly used functionality. Apart from the already mentioned parser library this includes libraries as simple as the IO library and as complex as a library that provides access to the system's call graph.

These libraries form a central point of entry to the analyzed system's artifacts and thereby allow the implementation of efficient caching strategies. As it is very likely that different processors will use e. g. the AST of a particular compilation unit the AST will be cached for future use and needs to be built only once. All *ConQAT* libraries use caching mechanisms which greatly reduces analysis time. To further improve performance the libraries are built on top of each other (if reasonable). The parser library uses pre-cached tokens from the lexer library. All caches are implemented in a memory-sensitive way and support dynamic uncaching if the system is short of memory.

4.2 Examples

Central goal of our endeavor was to create a tool that allows analysis and assessment of system properties that are really relevant in a given project situation. The following examples aim for illustrating how *ConQAT*'s flexibility supports very diverse quality assessment activities. The following excerpt of existing processors shows that it indeed supports a great variety of different analyses besides commonly known metrics like *lines of code* (LOC), *comment ratio* (CR), *coupling between objects* (CBO), etc.:

- *ANTRunner.* This processor executes a set of targets in ANT build files and monitors build success.
- *SVNLogMessageAuditor.* This processor checks if the commit log messages entered into the source code versioning system comply with the project guidelines.
- *JavaDocAuditor.* This processors checks if JavaDoc comments are present and comply to the guidelines.
- *JUnitRunner.* This processor runs unit tests and captures test outcome.
- *PMDRunner.* This processor acts as an adapter to the open-source tool kit PMD which offers a great number of source code audits.

Composition *ConQAT*'s architecture ensures that different processors can be flexibly composed. A simple example composition of two processors is a processor which determines the LOC for all source files and an aggregating processor which sums up these values to higher organizational levels like packages, subsystems or the whole system. Another more interesting processor could determine the number of methods per source file and calculate the average method length by using the LOC values above. Using a filtering processor these results could then be trimmed down to display the values only for a specific list of authors; authorship is determined by yet another processor via querying the source code management system. For example, this proved to be particularly useful in our practical labs as it allows us to concentrate on the results of certain students (or exclude all source files written by one of the teachers). If relevant, this information could then be cross-checked with unit tests results for the files these authors edited last.

Assessment As stressed before, the designated use case of *ConQAT* is not a single system analysis but the continuous assessment of quality criteria. To convey how *ConQAT* achieves this we give examples from our projects:

It proved to be useful to map project relevant data to a simple ordinal scale with the three values RED, YELLOW and GREEN. Although this may seem too coarse-grained for some applications, our experiences show that it is satisfying in most cases and most times better than a complex scale which is difficult to read and therefore mostly ignored. Apart from that, the ordinal scale and appropriate aggregation mechanisms prevent typical mistakes when calculating metric values [6]. Nevertheless *ConQAT* offers more complex options, too. If a situation demands for a more sophisticated assessment mechanism it can easily be realized.

For some time now we have used a similar three-valued scale in student projects to rate the review state of source code files: *premature*, *ready for review* and *accepted*. This rating information is stored in the source code files themselves. Obviously, we implemented a processor which extracts rating information from source files and maps them to the traffic light scale. Although it already proved to be handy to obtain a nicely formatted HTML output of the review state, we could really leverage *ConQAT*'s power by composing different analyses: With existing processors we could easily create assessments that e. g. check if all files rated as *accepted* have full JavaDoc documentation or comply to our coding guidelines. This also includes identification of typical errors like missing default cases in *switch*-statements and known anti-patterns like god classes. Files not complying to these rules are displayed with rating RED.

Using *ConQAT* we could also aggregate information to assess component- or system-specific criteria. Typical assessments check if percentage of *premature* files is below 10% or if a component (e. g. a package) exhibits any abnormal properties (as defined by the lower level rules).

4.3 Discussion

ConQAT's architecture proves to satisfy our requirements for flexible yet efficient quality analysis tools: It runs in a non-interactive manner and generates static HTML-output. It is flexible and extensible by offering two different levels of configuration; analyses can be composed using a declarative configuration file and new analyses can be added by implementing new processors. Lastly the system's design does not limit analyses to a particular scope, granularity, or type of artifacts.

An additional advantage of *ConQAT*'s architecture is the easy integration of new analyses. This enables users to leverage the abundance of already existing analysis tools. Especially the Java community offers an incredible rich variety of (open-source) analysis tools like parsers, coding convention checkers, test frameworks, etc.

ConQAT offers convincing performance characteristics due to the heavy use of caching mechanisms. Experiments show that analyses of systems as big as Eclipse ($\approx$ 2.5 MLOC[5]) can be carried out in a matter of a few minutes.

The downside of the tool's design is a fairly complex configuration mechanism. Setting up a complete quality assessment configuration results in a configuration file of about 300 lines and requires thorough understanding of the processors involved. However, the aim was to develop a tool which is configured rather infrequently and run repeatedly with the same configuration. Besides that we expect the typical user to be an expert in the field and therefore don't consider this issue a problematic one.

5 Conclusions

As a tool like *ConQAT* can't be an end in itself it's essential to ensure its usefulness in the context of continuous quality management processes. Up to now, we applied *ConQAT* during the development of a tool collection at our department. As the development team of this tool collection (students and researchers) is subject to rapid turnover and the tool collection amounting to 100.000 LOC, sound quality management is *the* crucial factor for productivity.

What we observed so far is that the application of the tool greatly improved the consciousness for quality aspects as part of daily development tasks. Students feel comfortable with getting an automated feedback on their work before they actually submit it and state that this system indeed motivated them to create quality code from the beginning.

[5] million lines of code

Noticeable, it was also the first time that we heard students intensively discussing code quality issues in the lab.

While we can't quantify this statement, we are sure that this quality controlling process pays off rapidly, even for small projects (e.g. 6 students, part-time, 4 weeks, 20.000 LOC).

Naturally, *ConQAT* is far from being complete. Currently the top-most item on our agenda is adding database support to store analysis and assessment results. This would allow us to track evolution of the system's quality more conveniently. Besides that we plan to carry out a controlled experiment to quantitatively measure the effects of continuous quality management during the next lab courses.

References

[1] W. R. Bischofberger, J. Kühl, and S. Löffler. Sotograph - a pragmatic approach to source code architecture conformance checking. In *EWSA '04: Proceedings of the 1st European Workshop on Software Architecture*, pages 1–9. Springer, 2004.

[2] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[3] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit. *Characteristics of Software Quality*. North-Holland, 1978.

[4] F. Deißenböck and M. Pizka. Concise and consistent naming. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 97–106, Washington, DC, USA, 2005. IEEE Computer Society.

[5] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, Jan. 2001.

[6] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. Softw. Eng.*, 20(3), 1994.

[7] International Standard Organization. *ISO 9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use*, Dec. 1991.

[8] A. Ludwig. *RECODER Technical Manual*. http://recoder.sourceforge.net, 2001.

[9] T. M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996.

[10] J. S. Reel. Critical success factors in software projects. *IEEE Software*, 16(3):18–23, 1999.

[11] STSC. Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense, Mar. 1997.

Metamodel-driven Service Interoperability

Andreas Winter Jürgen Ebert

Institute for Software Technology
University of Koblenz
D-56070 Koblenz, Universitätsstraße 1
www.uni-koblenz.de/~winter/
mailto:(winter|ebert)@uni-koblenz.de

Abstract: Interoperability in service oriented environments is heavily influenced by the view that the cooperating services have on their data.

Using the term service for the abstract contract concluded between a service requestor and a service provider, three different data schemas have be identified, namely the requestor's schema, the provider's schema and the reference schema introduced by the service specification.

Metamodeling and schema transformation approaches from the area of model driven architecture can be used to define these schemas and their mappings as well as the appropriate transformations that have to be done to the data.

1 Motivation

Interoperability is the challenge of enabling software tools from different suppliers to share their functionality. Services provided by one tool are used by another one. Coupling different but interoperable tools allows to form integrated collaborative tool suites without re-inventing and re-implementing already existing software components.

The issue of integration has already been addressed in the ECMA Reference Model [EC93], where data integration, control integration, presentation integration, process integration and framework integration have been recognized as different aspects. In the context of service oriented architectures this reduces to data integration, which describes the ability to share data between services and control integration which focusses on the flexibility on combining functionalities provided by services.

XML [W3C04a] intends to provide mechanisms to define standardized data exchange notations. Further advances in component-technology, like CORBA [OMG04a], and web-technology, like web services [ACKM04], offer additional infrastructure to enable tools to collaborate dynamically.

Service interoperability requires multiple communication steps between collaborating partners to supply a service. These steps contain various functionality, like e. g. initializing communication, requesting computations, and exchanging data, repeatedly. Often, functionality of a service has to be invoked in a certain order, according to the needs of the collaborating tools. Different tools are usually based on different data structures. Due to these data incompatibilities, mediators are needed to synchronize these data [We96, Si00].

Thus, service interoperability requires a *contract* between the partners, including a *specification* of the provided functionality, a *communication protocol*, defining the invocation

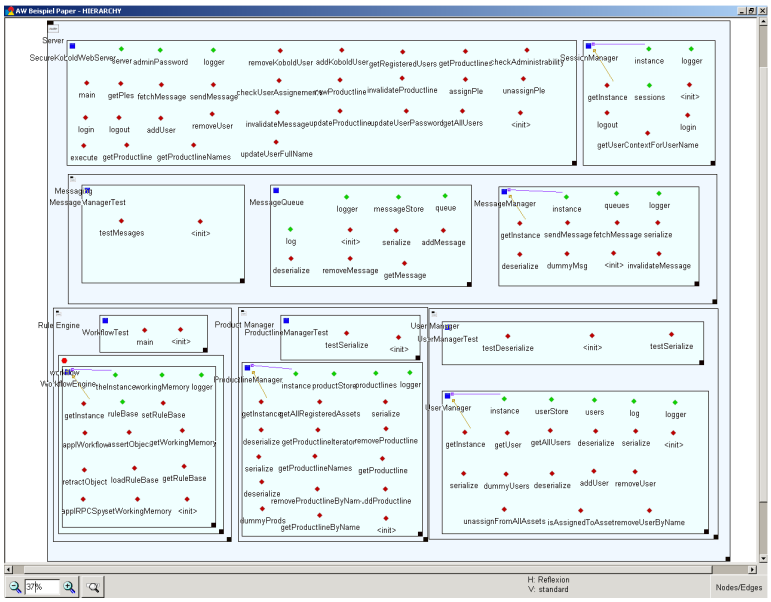


Figure 1: Decomposition View in Bauhaus

order, and a set of *reference schemas*, defining the data to be exchanged between collaborating services.

This paper focuses on the use of *metamodels* to adapt data between interoperable tools. Metamodels are used to define imported and exported data and found the base for transforming and exchanging data between collaborating tools.

Example. In the following, service interoperability of software *re*-engineering tools is used as an example. Similar experiences also hold for other domains. Reengineering-data is usually exchanged by GXL Graph eXchange Language [Wi02, HSEW05], which is accepted as the standard exchange language in reengineering [Dag01] and is supported by a broad variety of reengineering tools [GXL04].

Starting from the source code of a legacy software system, the *Bauhaus re-architecting tool* [CEK+00, EKB05] extracts a (possible) software architecture. The decomposition view [CBB+03] of a software architecture shows its structure in terms of components and subcomponents. The decomposition of an object oriented software system into packages, classes, attributes, and methods using the Bauhaus built-in visualization is presented in Figure 1. The following sections of this paper will show, how the introducing *metamodel-driven service interoperability* supports visualizing this architecture by means of UML class-diagrams [WW05].

Section 2 will introduce *services* and *components* and their major elements, which provide the base for service interoperability. The metamodel-based approach on service-interoperability is explained in section 3. A continuous example shows its realization according to the GXL metamodeling approach.

2 Services and Components

The shift from developing large monolithic systems towards service oriented architectures, including the potential of web service-based implementations, leads to new chances and challenges in software development. Instead of thinking of coupling *concrete software components*, interoperability is viewed as providing and using *abstract services*.

Services. Services form the interface to access functionality provided by *components*. They hide implementation details enabling service invocation regardless of specific hard- and software environments [Kr01].

We view *Services* as abstract descriptions of coherent functions with public interfaces [W3C04b]. They provide all information required to use the functions they describe. They contain at least *methods* (including their signature) for accessing the service's operations and/or *events* the service has to react on.

The functionality specified by a service, usually contains a set of operations. Each operation is specified by its *signature* defining input and output data. Operations have to be invoked in correct order. E. g. visualizing the software architecture shown in figure 1 has to succeed to an initialization step, which moves the required data to the layouter. The correct order of service interaction has to be specified in a *communication protocol*.

Interacting with services also requires to agree on the data used by the service to perform its functionality. *Service requestors* have to supply the input data in a specific form. Its notation is given by a *data interchange format* and its structure is specified in a *reference schema*, which is a common metamodel of the data.

Components. Services are implemented by components which are the *service providers*. There may be several alternative components implementing the same service. Service providers realize the service interface with all specified methods, events, communication protocols, and reference schemas.

Internal data structures of components, also specified by *schemas*, have to comply with the structure defined by the service's reference schema. Depending on the implementation of the provided operations, different service providers may use different internal implementations of specialized data structures optimized for efficient calculation.

The service's reference schemas are viewed as an *ontology* which specifys a shared conceptualization between the service and its components [He02]. It also serves a the data part of the *contract* between the service requestor, who has to deliver data according the reference schemas, and the service provider, who implements the service.

Example. In the example, we are looking for a *visualization service* offering functionality to import data on software architectures and to visualize this data by UML class diagrams. This service can be provided by any UML tool offering import and layout facilities. Here, we use the Rational Software Modeler [RSM] to collaborate with Bauhaus [EKB05]. Bauhaus supplies the re-architecting service (*ReArchService*) and the Software Modeler delivers the visualisation (*UMLService*).

3 Service Interoperability

Interoperability between services requires a *contract* on the operations supplied by the service provider and on the data and datastructures exchanged between the collaborators. The service contract is specified by the operation signatures, an interaction protocol and reference schemas. A possible interaction protocol, including the offered operations for connecting re-architecting and visualization services is sketched in Section 3.1.

Components usually employ internal data structures, that are optimized towards efficiency. Hence, collaborating components mostly do not use identical data structures and filtering of data is required. Section 3.2 shows how data filtering is realized by using the service reference schema as a shared conceptualization.

3.1 Protocol-based interaction

Correct interaction between collaborating partners is specified by protocols. The protocol of a "software architecture visualization" service has to ensure, that the service can only be invoked, if an extracted software decomposition is available.

Since multiple visualizations should be possible, the visualization service starts with importing the decomposition. Afterwards visualization operations can be invoked. Thus, the visualization service has to provide at least the operations `import` and `visualize`.

Figure 2 (black part) shows one possible interaction for visualizing the decomposition view of a software architecture by an UML service. Using re-architecting functionality, the decomposition view (*dv*) is extracted from an architecture according to the reference schema. This data is imported by the UML service and visualized afterwards.

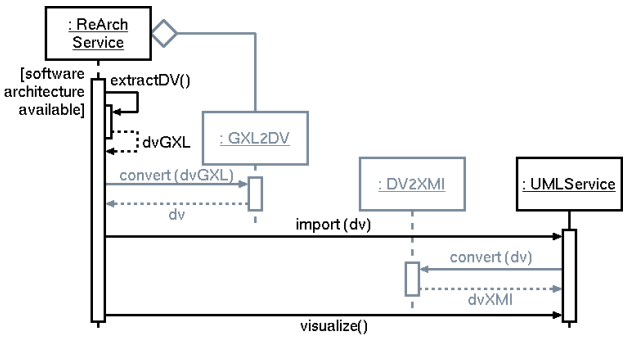


Figure 2: Interaction Scenario

Since service providers usually use different internal data structures, *filter services* have to be embraced. The scenario in Figure 2 (grey part) shows a *service configuration* which uses two additional services (*GXL2DV* and *DV2XMI*), which convert GXL (exported by the Bauhaus tool) to XMI (imported by Software Modeler). These filters can be realized in a service oriented architecture by operations within the components or by separate services. In the example the *GXL2DV*-filter is part of the ReArchService, whereas *DV2XMI* is a separate service [WW05].

3.2 Metamodel-based data exchange

Interoperable components have to agree upon exchanged data. Common *reference schemas* supply means to accommodate client data to the data required by service provider, and vice versa. Service interoperability necessitates powerful and adaptable mechanisms to *define*, *transform*, and *exchange* such data.

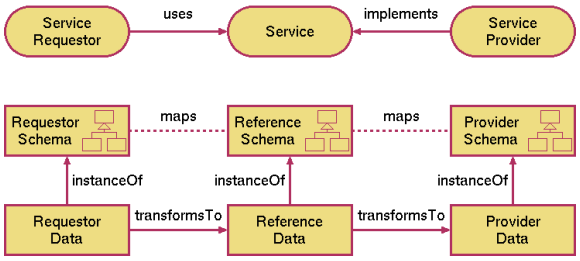


Figure 3: Metamodel-driven approach to Service Interoperability

Metamodeling provides such mechanisms. Metamodels define the structure of data to be exchanged. The reference schema, defining the services data, and the schemas used by the service requestor and the service provider offer the formal base to assign data transformations. Combined with a notation for packing data and schemas, metamodels also define an exchange notation. Figure 3 sketches this approach.

Data exchange in service oriented environments follows a stepwise approach:

1. **Define schemas**
   (a) Define reference schemas
   The reference schemas of the service serve as domain ontology reflecting the shared concepts among the collaborating components. They build an intermediate between interoperable partners.
   (b) Define service requestor and service provider schema
   Collaborating partners work on their own data structures. These have to made explicit [JCD02] to form the base of data export and import.
2. **Specify data transformations**
   Data according to the participating schemas have to be mapped on each other. Mappings from the service requestor data via the reference schema to the service provider data have to be specified. A nearly loss-free interchange can only be provided if the specified schemas only differ slightly. Possible loss of precission in discussed e. g. in [CJ05].
3. **Exchange transformed data**
   Data has to be exchanged in a standardized form, which conforms to the reference schemas. Since different services use different reference schemas, data has to be exchanged together with its corresponding schema information.

The defined schemas establish a base for both, *data transformation* and *data exchange*. The following sections show the schemas involved in realizing a software architecture visualization service, with Bauhaus and Software Modeler as cooperating services.

3.2.1 Defining Schemas

A service for visualizing the decomposition view of object oriented software architectures requires to agree on the data to be displayed. Figure 4 defines a possible decomposition of object oriented software systems. *Packages* contain further *packages* and/or *classes*. *Classes* contain *methods* and *attributes*.

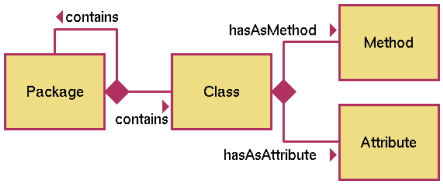
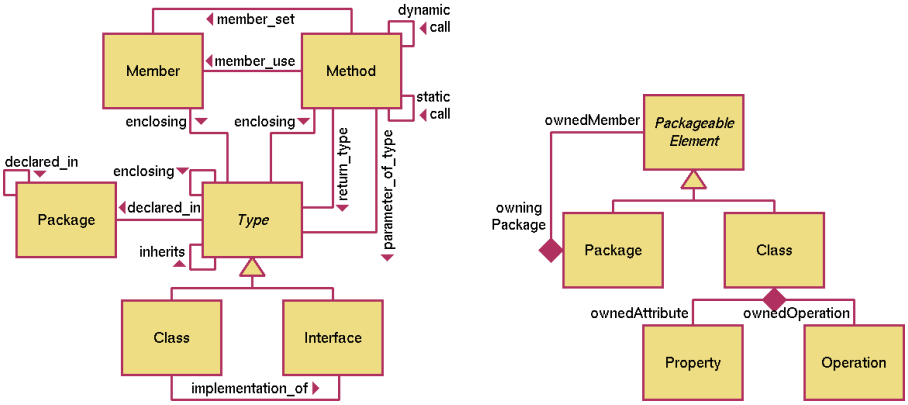


Figure 4: Reference Schema

Together with the protocol and the signatures the reference schema in Figure 4 defines the contract each service provider has to fulfill. A service requestor can be certain, that these data are visualized correctly by a service conformant provider.

Using services requires to map the data used by the service requestor to the reference schema. The data structure used by the Bauhaus re-architecting tool is shown in Figure 5a [CEK+00, EKB05]. *Packages* declare *classes* and *interfaces*. Both consist of *members* and *methods*. Bauhaus also represents further relationships between components of software architectures, not considered by our visualization service.



(a) Bauhaus Schema (b) UML Metaschema (extract)

Figure 5: Participating Schemas

UML tools usually found on the UML 2 superstructure metamodel [OMG04b]. Figure 5b shows an extract of the UML metamodel representing the relevant concepts for visualizing the decomposition view. Here *classes* contain *properties* and *operations*. *Packages* and *classes* are subsumed to *Packageable Elements* which can be substructured into further *packages* and *classes*. Software Modeler provides import facilities for data according this schema, based on XMI [OMG02a].

Apparently, all three schemas are capable to describe the decomposition of a software system into packages, classes, attributes, and methods. But, they use different terminology and modeling styles. The internal schemas of the collaborators are allowed to represent further data. So, e. g. the *Bauhaus schema* also supports inheritance and call relationships and the (complete) *UML schema* supports all other UML notations.

The *reference schema* serves as an intermediate, normalizing these structures. It constitutes a consistent terminology and defines a common conceptualization required for sharing re-architecting and visualization facilities between the collaborators.

3.2.2 Transforming Data

To enable data exchange between collaborating tools, data exported by the service requestor has to be adapted to the specific import format required by the selected service provider. The common conceptualization of both tools is specified in the services reference schema. Thus, two *transformations* are required: one transforms the requestors data to data according the reference schema, the second transformation moves data according the reference schema into the providers notation. If data has to be mirrored back, proper inverse transformations are needed.

Model transformation are a core constituent of model-driven architecture [OMG03]. Model transformation is viewed as the process of converting models into other models. Those transformations are specified in terms of *mappings* between the corresponding metamodels. Various metamodel based approaches are currently under development to enable specification and execution of metamodel based transformations [OMG02b].

The required mappings between the Bauhaus schema (cf. Figure 5a), the reference schema (cf. Figure 4), and the UML metaschema (cf. Figure 5b) are sketched in Figure 6. The mapping has to define the correspondence between the participating concepts and their relationships.

Bauhaus Schema	Reference Schema	UML Metaschema
p,q : Package *c : Class* *a : Member* *m : Method*	*p,q : Package* *c : Class* *a : Attribute* *m : Method*	*p,q : Package* *c : Class* *a : Property* *m : Operation*
$p \leftarrow_{declared\_in} q$ $p \leftarrow_{declared\_in} c$ $c \rightarrow^{*}_{inherits} \leftarrow_{enclosing} a$ $c \rightarrow^{*}_{inherits} \leftarrow_{enclosing} m$	$p \rightarrow_{contains} q$ $p \rightarrow_{contains} c$ $c \rightarrow_{hasAsAttribute} a$ $c \rightarrow_{hasAsMethod} m$	$p.ownedMember \ni q$ $p.ownedMember \ni c$ $p.ownedAttribute \ni a$ $p.ownedOperation \ni m$

Figure 6: Mapping between corresponding concepts and relations

The participating schemas in the architecture visualization example are very similar, only minor changes in names and directions of associations occur. The reference schema in Figure 4 does not support generalization on classes, whereas the Bauhaus schema in Figure 5a does. The mapping in Figure 6 takes this into account by assigning all members and methods of superclasses to subclasses.

Interoperation scenarios with very diverse metaschemas might require more complicated constraints specifying the mapping. In particular, complex structured associations between certain objects have to be considered.

Bauhaus, Software Modeler and the GXL meta modeling approach provide XML based import and export facilities. In [WW05] transformations were realized by XSLT scripts.

XSLT works rather inefficiently on huge amounts of data, which occur in interoperable reengineering environments. Furthermore, metamodels and their mappings do not influence the XSLT transformation explicitly. Only the source model appears in patterns, the target model is ignored. Thus, more efficient and powerful declarative metamodel-driven approaches are needed for model transformation. More promissing approaches for metamodel-based transformation might be BOTL [BM03] and MDI [KS05].

3.2.3 Exchanging Data and Schemas

GXL (Graph eXchange Language) [HSEW05] is an XML-based standard exchange format for sharing graph data between tools. GXL comes with a metamodeling approach, such that graph-based data can be exchanged together with its corresponding schema information. In contrast to other XML-based metamodeling approaches, GXL is homogeneously graph-based on all meta-levels, leading to one single and simple notation for instance data, schemas and metaschemas.

In the example, the Bauhaus schema as well as the reference schema are defined by GXL schemas, i. e. they can be exchanged as graphs corresponding to the GXL metaschema. So, the mapping between Bauhaus data and the reference schema was directly done by exchanging and transferring GXL documents. Since Software Modeler only supports XMI, a separate filter service was required. Data according to the reference schema were transformed into the XMI imported by the modeler.

Note, that e. g. SOAP [W3C03] also allows the inclusion of a schema in message bodies using the XML namespace mechanism. Though the meta-modeling power of XML is not very strong, in principle the statements of this paper might also be applied to SOAP.

4 Conclusion

This paper introduced *metamodel-driven service interoperability*. Interoperability between service provider and service requestor is viewed as fulfilling a contract defined by the service. Instead of coupling concrete software components individually, *service interoperability* views tool integration on an abstract level. Services specify the offered *functionality*, the *communication protocol*, and a set of *reference schemas*. Collaborating partners have to agree on this service specification.

Within this framework, data exchange plays an important role. Due to different internal data structures, transformations between collaborating tools are required. Data exchange and the transformations are realized according to the explicit schemas used by the tools and to the reference schema of the services, defining a common conceptualization.

Figure 7 shows the resulting UML visualization of the decomposition view from Figure 1 from the running example. Based on the specification of a visualization service the Bauhaus re-architecting tool was able to use the visualization functionality provided by the Software Modeler.

Both tools, Bauhaus and Software Modeler are not implemented with a strong service oriented architecture, which required to add additional filter services. But, currently approaches on migrating systems towards service oriented architectures (e. g. [KZ04]) are

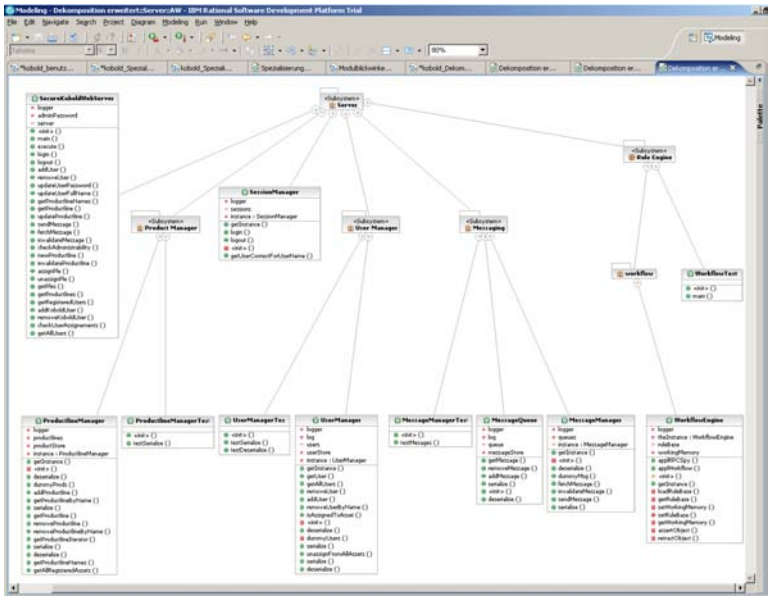


Figure 7: Decomposition View visualized with Software Modeler

being developed, probably leading to a stronger coupling of these services.

The presented approach is independent from concrete realization strategies. Visualizing a recovered software architecture and presenting the results in an UML tool was done with monolithic tools offering export and import facilities and some additional scripts. In a similar (and easier) way, the approach works in distributed environments based on web services. Similarly, the applied metamodeling approach is not restricted to GXL only. But, it becomes important, to adopt powerful *metamodel based transformation techniques*.

Acknowledgments: This proposal has benefited from various discussions with our students. We thank Kerstin Falkowski, Alexander Kaczmarek, and Julia Wolff for their collaboration, insights and suggestions.

References

[ACKM04] Alonso, G., Casati, F., Kuno, H., and Machiraju, V.: *Web Services, Concepts, Architectures and Applications*. Springer. Berlin. 2004.

[BM03] Braun, P. and Marschall, F.: Transforming Object Oriented Models with BOTL. *ENTCS*. 72(3):1–15. 2003.

[CBB+03] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J.: *Documenting Software Architectures*. Addison Wesley. Boston. 2003.

[CEK+00] Czeranski, J., Eisenbarth, T., Kienle, H., Koschke, R., and Simon, D.: Analyzing xfig Using the Bauhaus Tool. In: *WCRE 2000*. IEEE. Los Alamitos. pp. 197–199. 2000.

[CJ05] Cordy, J. and Jin, D.: Factbase Filtering Issues in an Ontology-Based Reverse Engineering Tool Integration System. In: *J.-M. Favre, M. Godfrey, A. Winter (eds.):2nd International Workshop on Meta-Models and Schemas for Reverse Engineering. ENTCS*. to appear. 2005.

[Dag01] J. Ebert, K. Kontogiannis, J. Mylopoulos: Interoperability of Reverse Engineering Tools. http://www.dagstuhl.de/DATA/Reports/01041/). 2001.

[EC93] ECMA: Reference Model for Frameworks of Software Engineering Environments, Version 2.6. Technical Report ECMA TR/55. European Computer Manufacturers Association. 1993.

[EKB05] Eisenbarth, T., Koschke, R., and Bellon, S. Bauhaus Software Technologies, Language Guide 5.1.4. Documentation. 28. April 2005.

[GXL04] GXL: Graph Exchange Language. http://www.gupro.de/GXL/tools/tools.html. 2004.

[He02] Hesse, W.: Ontologie(n). *Informatik Spektrum*. 16(6):477–480. Dezember 2002.

[HSEW05] Holt, R. C., Schürr, A., Elliott Sim, S., and Winter, A.: GXL: A Graph-Based Standard Exchange Format for Reengineering. *Science of Computer Programming*. (to appear). 2005.

[JCD02] Jin, D., Cordy, J. R., and Dean, T. R.: Where's the Schema? A Taxonomy of Patterns for Software Exchange. In: IWPC 2002. IEEE. Los Alamitos. pp. 65–74. 2002.

[Kr01] Kreger, H. Web Services, Conceptual Architecture (WSCA 1.0). http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf IBM Software Group. May 2001.

[KS05] Königs, A. and Schürr, A.: Multi-Domain Integration with MOF and extended Triple Graph Grammars, http://drops.dagstuhl.de/opus/volltexte/2005/22/pdf/04101.KoenigsAlexander1.Paper.pdf. In: Bezivin, J. and Heckel, R. (Eds.), *Language Engineering for Model-Driven Software Development*. Dagstuhl Seminar Proceedings 04101. 2005.

[KZ04] Kontogiannis, K. and Zou, Y.: Reengineering Legacy Systems Towards Web Environments. In: *K. M. Khan, Y. Zheng (eds.): Managing Corporate Information Systems Evolution and Maintenance, Idea Group Publishing, Hershey*. pp. 138–146. 2004.

[OMG02a] OMG XML Metadata Interchange (XMI) Specification. http://www.w3.org/TR/2000/REC-xml-20001006.pdf. January 2002.

[OMG02b] Request for Proposal: MOF 2.0 Query/Views/Transformations RFP. http://www.omg.org/docs/ad/02-04-10.pdf. October 2002.

[OMG03] MDA Guide. http://www.omg.org/docs/omg/03-06-01.pdf. June 2003.

[OMG04a] Common Object Request Broker Architecture: Core Specification. http://www.omg.org/docs/formal/04-03-01.pdf. March 2004.

[OMG04b] UML 2.0 Superstructure Specification. http://www.omg.org/docs/ptc/04-10-02.pdf. October 2004.

[RSM] IBM Rational Software Modeler. http://www-306.ibm.com/software/awdtools/modeler/swmodeler/.

[Si00] Sim, S. E.: Next Generation Data Interchange, Tool-to-Tool Application Program Interfaces. In: *WCRE 2000*. IEEE. Los Alamitos. pp. 278–280. 2000.

[W3C03] SOAP Version 1.2 Part 0: Primer, W3C Recommendation. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/. June 2003.

[W3C04a] Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation. W3C XML Working Group. http://www.w3.org/TR/2004/REC-xml-20040204. February 2004.

[W3C04b] Web Services Architecture. W3C Working Group Note. W3C Web Services Architecture Working Group,. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/wsa.pdf. February 2004.

[We96] Wegner, P.: Interoperability. *ACM Computing Survey*. 28(1):285–287. 1996.

[Wi02] Winter, A.: Exchanging Graphs with GXL. In: Mutzel, P., Jünger, M., and Leipert, S. (Eds.): *GD 2001*. *LNCS 2265*. Springer. Berlin. pp. 485–500. 2002.

[WW05] Wolff, J. and Winter, A.: Blickwinkelgesteuerte Transformation von Bauhaus-Graphen nach UML. *Softwaretechnik-Trends*. 25(2):33–34. Mai 2005.

Stub Libraries for Software Migration and Development (Position Paper)

Pradeep Varma
IBM India Research Laboratory
Block 1, Indian Institute of Technology
Hauz Khas, New Delhi 110016, India
+91-11-26861100, 26861555 (FAX)
pvarma@in.ibm.com

ABSTRACT

We propose stub software libraries for testing programs on platforms on which the regular libraries may not be available. This may be encountered during outsourced software development (libraries unavailable to outsourcing provider), and development/migration in advance of vendor-provided libraries. Briefly, a stub library comprises a cached front-end component (including package headers) for integrating with the client program in standard fashion, with calls to the library either being served by locally-cached answers in the front-end, or by remote invocations of a back-end on a platform supporting a live image of the library (if available). The ability to cache library request-response pairs implies the method can be used to inventory usage of libraries as a part of the standard inventory process in an outsourced porting/migration/development engagement.

For uniform treatment of diverse datatypes and a common shared interface with the optional remote backend, our cache construction represents all data in marshalled form. We describe variations of the caching method, including predetermined set of library calls, dynamically determined set of library calls, and varying degrees of cached results and cache revamping for the same. Rerouting library calls to stub libraries semi-automatically may be carried out by library calls analysis in application sources using syntax and approximate pointer analysis in the C/C++ context. Dynamic checks embedded in the stub libraries then ensure safety against false negatives present in the approximate results.

OVERVIEW

We describe a testing tool by which approximations of runtime libraries can be made available on platforms on which the actual libraries themselves are not available. For example, in an outsourced porting scenario, the libraries may not be available on the target platform to which the software being ported is headed. In *ab initio* software development, or in software re-engineering purposes, the development of the software may precede actual library development, necessitating usage of approximations as described here for testing purposes.

In the context of software development/migration by source-to-source transformations, the transformation toolkit can often get by solely using a compiler front-end as in [1, 2]. Testing support however necessarily requires the ability to fully compile the application being ported and to link and execute it on at least a virtualized run-time representing an actual environment. In the present work, we describe virtualized libraries, i.e. stub libraries, as testing support. Support needs for source-to-source transformations are as shown in figure 1. In figure 1, compilation processes typically require compiler front-end and library headers to verify correct compilation. There may be exceptions, as for instance the presence of build flags requiring intermediate code optimizer may force a presence of middle compiler components also for transformation verification. On the other hand, testing support always requires the full compiler (at least an emulation thereof), and (stub) libraries. Headers alone do not suffice since library calls have to actually be executed by minimally a virtualized library.

In general, during software development, the availability of libraries cannot be presumed. Our work tackles this requirement by generalizing the notion of software inventory preceding development activity to include a library-usage inventory step, whereby actual or anticipated calls to libraries for a given set of input scenarios (test cases) are collected and cached for re-use during development. While our work is not limited to a given language context, we describe it in the general C/C++ context for specificity. Indeed for purely functional languages the problem is much simplified [3] and cache applicability to library calls is far more relevant than the case in commercial, "imperative" languages like C/C++. Regardless, our work applies to diverse languages, including functional, imperative, and non-deterministic languages and we use a difficult, C/C++ context to drive its discussion. Our discussion covers a semi-automatic method of building stub libraries that are specific to usage of a given application context. This includes syntax and approximate pointer analysis by walking the application sources syntax tree to determine the potential set of library calls embedded in the application code. Based on name bindings and type signatures [4], a general set is identified that is a reasonable, but not guaranteed superset of all library usage. For non-standard invocations that might escape such use classification, a dynamic safety test is embedded in library functions that are not converted into stub versions. This suffices to guarantee safety against false negatives in the approximate information collected for identifying application-specific use of libraries.

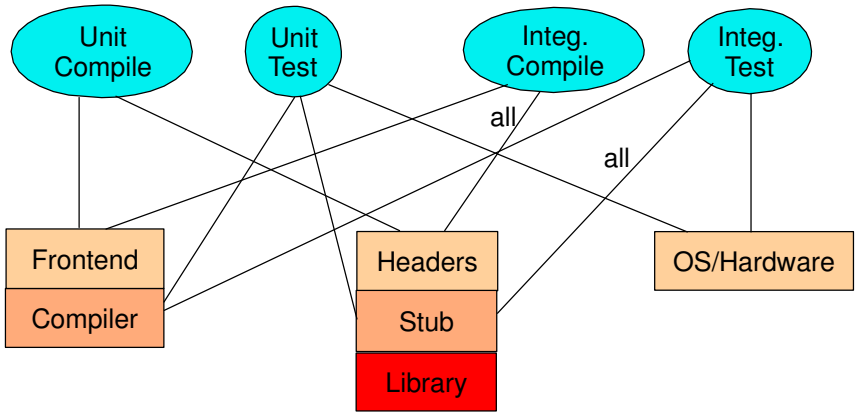


Figure 1. Minimal support requirement of individual practices. Shifting lower implies more extensive support need.

Figure 2 shows the general schematic of our stub library. A client program is shown linking to a library package's header. The usual header is substituted by the header shown in blue, by the use of which, the object code vector shown in red is linked to. A library function `abc()` in source code is shown linking to `abc1()` in the stub front-end thus. Calls to `abc1()` first (optionally) refer to a front-end cache, followed by (optional) interprocess/distributed communication to the back-end image. For the back-end image, two vectors are shown, one showing the wrapper code (yellow) and the other showing the actual library. Modified header files for building the front-end and back-end code vectors are not shown since the build process for the vectors follows usual methods.

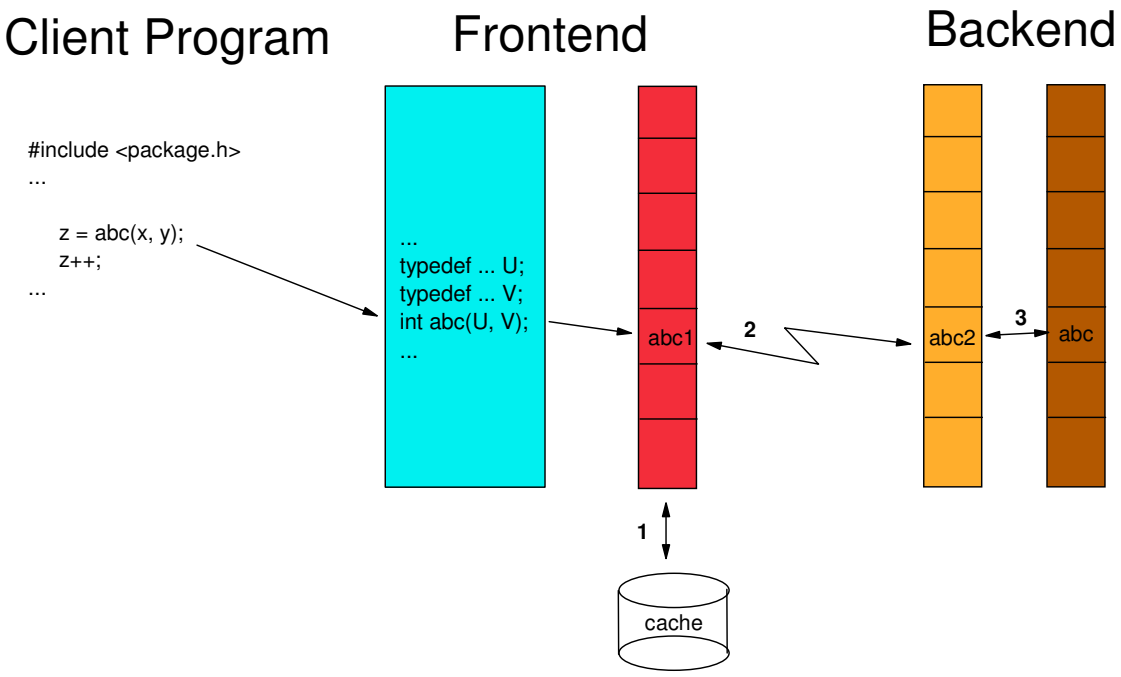


Figure 2. Stub Library Architecture

The stub functions built by our system rely on standard marshalled representation of call arguments as the cache index by default. The same representation is used for communication with the remote backend, whenever applicable. Performance overheads can be reduced by relying upon a specialised (partial evaluated) cache representation, one per function, at the cost of increased space management complexity among functions. In comparison to related work, where cached/memo functions are well known (especially. in referentially-transparent, purely functional languages [3]), the design of library substitutes for "real-world" languages using a combination of optional cache functions supported by an optional live back-end on an independent platform is not known.

Use of the stubs can invoke any combination of the optional cache and optional back-end components. For cache alone to be invoked, the set of calls has to be predetermined, the collation of which prior to stub invocation can be carried out by appropriate tracking of library

calls in the original source environment of the software being ported/developed. For back-end alone to be invoked, the front-end serves solely as a conduit to the back-end and caches no results in the interim (e.g. the functions being invoked are non-deterministic and hence caching is not sought). For both the cache and back-end to be invoked, the set of calls kept cached can be either static or dynamic, with the static case covering a predetermined set of calls. Regardless of the optional cache use or not, the front-end interfaces with the client program as substitute headers files that link to substitute wrapper functions which invoke the combination of cache/back-end needed to respond to a regular call. Cache use may be carried out conservatively/safely in calls to pure functions, or references to immutable data and type entities since these calls and references return the same answer each time. Functions that are non-deterministic (e.g. time, date) need to be verified individually whether a (non side-effecting) caching approximation for the functions can suffice or not. For non-deterministic functions (e.g. coin flip), if for the temporary purposes of migration intermediates or initial development, it is enough to use the first value returned by the function as a stand-in answer, then a cached version of the function can be used. While standard compiler techniques can automatically verify whether a function is pure or not in some cases, in general, decision-making for whether a given function can have a stub substitute has to be made manually. We will discuss these as well as a simple pointer and syntax analysis for reducing stub support to only those functions that may be exercised by user code. Our analysis is closest to the (exact) signature matching technique suggested by [4] for the purpose of navigating or browsing a library, locating functions and modules.

REFERENCES

[1] Devanbu, P. T. GENOA – A Customizable, Front-End-Retargetable Source Code Analysis Framework. *ACM Trans. Soft. Eng. Method., Vol. 8, No. 2,* (April 1999), 177-212.

[2] Varma, P., Anand, A., Pazel, D. P., and Tibbitts, B. R. NextGen EXtreme Porting: Structured by Automation, In *Proceedings of the ACM Symposium on Applied Computing (SAC 2005)* (Santa Fe, NM, USA, March '05) ACM Press, New York. 1511-1517.

[3] Varma, P., and Hudak, P. "Memo-functions in Alfl", Research Report YALEU/DCS/RR-759, Yale University, December 1989.

[4] Zaremski, A. M., and Wing, J. M. Signature Matching: A Tool for Using Software Libraries. ACM Transactions on Software Engineering and Methodology, Volume 4, Number 2, (April 1995), 146-170.

Applying Software Transformation Techniques to Security Testing

Songtao Zhang, Thomas Dean
Electrical and Computer Engineering
Queen's University
4sz7@qlink.queensu.ca,dean@cs.queensu.ca

Scott Knight
Electrical and Computer Engineering
Royal Military College of Canada
knight-s@rmc.ca

Abstract

Application protocols have become sophisticated enough that they have become languages in their own right. At the best of times, these protocols are difficult to implement correctly. Combining the complexity of these protocols with other development pressures such as time to market, limited processor power and/or demanding performance requirements make it even more difficult to produce implementations without security vulnerabilities. Traditional conformance testing of these implementations does not reveal many security vulnerabilities. In this paper we describe ongoing research where software transformation and program comprehension techniques are used to to assist in the security testing of network applications. In particular, security testing of state stensitive protocols is described

1. Introduction

The security of network applications is an increasingly important topic in both academia and industry. The cheap availability of bandwidth world wide has increased the ability of people to communicate, but has also provided convenient access to many systems for those with malicious intent. Additionally, implementations formerly on closed network protocols are moving to public protocols such as the move of telephone networks from packet switched networks to Voice over IP protocols.

Conformance testing of these applications tends to focus on the correct implementation of the application to valid requests and obvious errors. The protocols used by network applications have become languages in their own right with syntax and semantics. One approach to security testing is Syntax Testing [1]. In this approach, syntax and semantic errors are intentionally made to produce variants of the data to attempt to expose vulnerabilities.

The PROTOS project[7] at Oulu University uses a protocol grammar using higher order attribute grammars to generate variant packets. The grammar specifies the possible packets right down to the values of fields. The grammar is modified using a script to allow the desired errors and then a walker walks the grammar tree, automatically generating the packets and analyzing responses. The higher ordered attribute grammar actions are custom written java routines. They are triggered when the walker reaches specific grammar nodes. Examples of actions include copying values from result packets to client packets (including aribtrary computation such as password encryption) and computing checksums of packets.

While the general technique is similar, our approach is different. We capture a valid set of data by sniffing the network and transforming it to generate alternate packets. We also are automatically generating the test plans based on the syntax and semantics of the protocol without manual intervention. This paper describes our approach to handling state based protocols. That is protocols where the values of one packet depend on the values of previous packets.

2. System Structure

Figure 1 shows the overall structure of our system. At the bottom of the figure we have a network containing the test system and a client system that is interacting with the test system. A sniffer is used to capture one or more valid protocol data units (PDUs) that are sent from the client to the test system. A PDU may be a single packet, or it may be spread over multiple packets. The captured PDUs at this point in time are stored in binary data files. These files are decoded into a textual representation by a decoder.

The markup and execution engine, implemented in TXL[3], are used to generate variants of the PDUs which are then re-encoded and injected back into the network. The markup and execution approach is modeled on previous research [4]. This approach separates the planning of the testing suite from the execution of the testing suite. The markup that is generated is rather simple. It includes markup to delete a field, change the encoding of a field, duplicate a field, change the value of a field, and other similar tasks.

The execution engine carries out most of the markup (encoding markup is carried out by the encoder). Thus markup is always done on the original valid data, and may generate more than one PDU, while execution

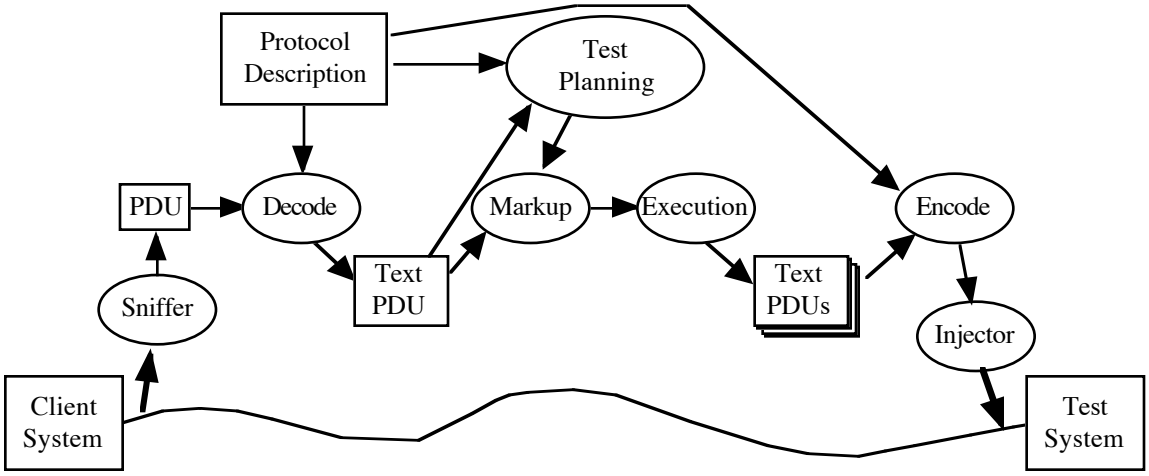


Figure 1. Protocol Tester General Structure

generates the modified PDUs. This separation of concerns is important. Testing strategies that depend on simultaneous changes to multiple fields communicate through the markup. That is, they make markup to simultaneous fields. The execution engine, responsible for implementing the transforms need not know about relationships between fields.

The description of the protocol contains a variety of information. It contains the syntax of the protocol, transfer encoding information and semantic information such as constraints between fields, ordering of data fields and if a given field must be unique. Our approach extendeds the industry standard ASN.1[6] protocol description language with markup to provide information about semantic constraints in the notation.

3. Extending Protocol Tester for States

Our recent research [5,8] has focused on stateless protocols. That is request–response protocols where each request is independent of any previous requests. This includes protocols such as X.509[10,11], OSPF[9] and SNMP[2]. We have started to extend the framework to handle state based protocols. These are ones in which the value in one PDU is dependent on a value in a previous PDU. For example, in the SMB protocol, the server answers the initial negotiation request with a key string which must be combined with the users password to sucessfully log in to the server.

One approach would be to model the protocol using automota, which are used in some cases to specify the protocol. However, this is a heavy weight solution in that the interpreted state machine is essentially a functional client. PROTOS uses a script and actions in a higher order attribute grammar. The exact exchange to be tested must be scripted using a combination of the grammar and actions. Our insight is that why not let an actual client drive the sequencing of the data units?

As with the stateless version of protocol tester, we capture the PDUs by sniffing the network. However this time we capture the sequence of PDUs including the replies from the test system. Each packet is decoded into its textual form, and we choose one or more of the PDUs in the sequence to mutate. We can then replay the sequence of data up until the mutated PDU and determine the response of the test system.

However, replaying the data is not as simple as simply retransmitting the binary form of the PDUs. Some information is dependent on information that is returned by the server, specifically to deny replay attacks against the system. We have a designed a scripting langauge used to drive the injector that can copy data between packets and perform limited computations. Figure 2 shows an example of this scripting language.

The example is for an exchange with a Samba server. The client logs into the server, and reads a file from the server. The client sends a total of 9 PDUs to the server, with a response for each. Most of the PDUs are independent (other than in time), and there are only three state dependent relationships between packets in the sequence.

The client initially sends a negotiation packet, listing the variants of the SMB protocol that are understood by the client. The server responds with a choice of variant, a session key and a challenge string. The client then sends a login packet with the session key and a value computed from the users password and the challenge string. In figure 2 the first two lines of the script describe the password management. The first line records the password for use (“zhang”), This is provided by the

```
password == "zhang";
sendPK.2.65.88 == SMBencrypt(password, recPK.1.73.80);
sendPK.2.47.51 == recPK.1.52.56;
sendPK.6.37.38 == recPK.5.37.38
```

Figure 2. Injector Script for a SMB PDU Sequence.

test engineer. The second line states that bytes 65 through 88 of the second packet are computed using the SMBencrypt function (built into the injector), the password and the contents of bytes 73 thrrough 80 of the first response (**rec**eived) PDU. The third line indicates that the session key should be copied from bytes 52 trhgouh 56 of the first response PDU to bytes 47 through 51 of the second PDU to be sent by the injector.

The last line of the script indicates that the file identifier should be copied from the response PDU to the open PDU (the 5th PDU) to the Read PDU (the 6th PDU).

The script will be automatically generated from the protocol description. We have extended SCL(which was is an extension ASN.1) to include dependencies between PDUs as well as between fields within a PDU. The original constraints between fields within the PDU indicate fields that should be tested by the mutation engine. These constraints between packets indicate where the injector should copy data, possibley computing a value. A set of predefined functions are provided by the injector such as calculating checksums and encrypting passwords.

Since some of the fields may have variable lengths and to make the specification simple to author, The constraints are specified by field name. Thus the fourth constraint would be specified in SCL as "ReadCmd.FileNum == OpenResp.FileNum". The captured PDUs are decoded and the offsets in the captured PDUs are used when generating the test script. Since PDU 5 is an Open Command PDU, and PDU 6 is a Read PDU, the constraint is translated to the version in Fig. 2. Thus, a given test script is only valid for given set of captured PDUs. A new test script can be automatically generated from a new set of captured PDUs.

4. Future Work

The system we have described is a very general infrastructure with a great deal of potential. Some of the future work we are planning on pursuing include:

The protocols described are also currently binary protocols. Textual protocols such as HTTP, SMTP and SOAP (XML over HTTP) can also be security tested using a transformation based process. The twist is that interesting servers to test are multi tired.

The current approach is also based on a black box approach. On some occasions when we have had source code to the test system, we have tracked down the bug in the system manually. Expanding to a white box style of testing has some potential. One option is to use the erroneous PDU to isolate the error automatically. The other option is to use a light weight program comprehension/design recovery step to identify potential security failures in the system. A full comprehension approach can be expensive both in time and resources. A light weight identification could be more aggressive in identifying potential vulnerabilities which are used to provide information to the test planner.

5. Conclusions

This paper has presented some of the ongoing research into network security at the Royal Military College of Canada and Queen's University. The key contribution in our approach is to avoid haveing to specify the state dependencies either as an autonoma or as a test script or grammar. We need only specify the data dependencies in a declarative format, and a working client will provide us with the state transfition information necessary to run the test.

References

[1] Bezier, B., *Software Testing Techniques*, 2nd Edition, Van Nostraad Reinhold, New York, 1990.

[2] Case, J., Fedor, M., Schoffstall, M., Davin, J., Simple Network Management Protocol, Internet RFC 1157, 1990.

[3] Cordy, J., The TXL Programming Language, v. 10, http://www.txl.ca/, 2000.

[4] Dean, T.R., Cordy, J.R., Schneider, K.A., Malton, A.J., "Using Design Recovery Techniques to Transform Legacy Systems", *ICSM 2001 - The International Conference on Software Maintenance*, Florence, Italy, November 2001, pp 622 - 631.

[5] Dean, T.R. Knight, G.S.N, "Applying Software Transformation Techniques to Security Testing", *International Workshop on Software Evolution and Transformation*, Delft, Netherlands, November 2004, pp 49-52

[6] Dubuisson, O., "ASN.1 Communication between Heterogeneous Systems", Academic

Press, San Diego, 2001.
[7] Kaksonen, R., Laasko, M. and Takanen, A., *Vulnerability Analysis of Software through Syntax Testing*,http://www.ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/index.html, 2001.
[8] Marquis, S, Dean T., Knight, G.S.N, SCL: A Language for Security Testing of Network Applications, *Proc. CASCON 2005*, Toronto, Oct. 2005. to appear.
[9] Moy, J., OSPF Version 2, Internet RFC 2328, 1998.
[10] PKCS#7-Cryptographic Message Syntax Standard. http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/, RSA Data Security, Inc, 2004.
[11] Public-Key Infrastructure (X.509) (pkix). http://www.ietf.org/html.charters/pkix-charter.html, 2004.

Workshop 3.1:
Workshop on Net-Centric Computing (NCC 2005):
Middleware -- The Next Generation

Knowledge Sharing in Multi-Layer P2P Networks

Paraskevi Raftopoulou, Euripides G.M. Petrakis
Department of Electronic and Computer Engineering
Technical University of Crete (TUC)
Chania, Crete, GR-73100, Greece
{paraskevi,petrakis}@intelligence.tuc.gr

Abstract

We present an information sharing architecture based on ideas from semantic information retrieval, structured overlay networks and recent work on p2p systems. The architecture aims at providing easy access to information by exploiting the various aspects of knowledge characterizing information resources in a p2p network. Different aspects of knowledge are organized into similarity graphs and are clustered separately in different layers so that, peers do not only explicitly point to one another but also, they are implicitly related by virtue of containing related information resources. Queries are resolved by addressing each layer independently and by combining results obtained from each layer.

1 Introduction

Knowledge sharing aims towards real-time, low overhead and dynamic information sharing and dissemination within user communities (e.g., members of an organization) sharing common interests or goals. The introduction of *peer-to-peer* (p2p) paradigm [2] to knowledge sharing in user communities is currently seen as a promising technology to overcome the challenges. Structured overlay networks (e.g., [1, 6]) are also being introduced as tools for better organization of information and for sharing of different aspects of knowledge residing in a network of peers.

We present a knowledge sharing model and a multi-layer architecture based on ideas from information retrieval, structured overlay networks [1] and recent work on p2p systems [4]. The proposed architecture is expected to serve *ad-hoc querying* functionality. The main components of this architecture are *information providers* (e.g., the available data sources) and *information consumers* (e.g., users querying for information). A peer can stand for both an information provider and an information consumer.

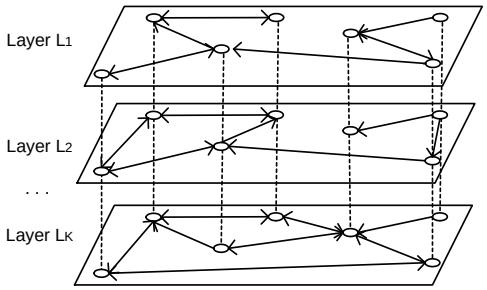


Fig. 1: Multi-layer graph model.

Various aspects of information either specific to data type (e.g., music, video, science), data representation (e.g., schema information) or users (e.g., user or peer profiles) indicate different structure layers. Data resources are clustered at each layer and appropriate lookup functions are employed to locate the relevant data items within each cluster and within each layer separately. Data sources on each layer can explicitly point to one another, or they may be implicitly related by virtue of containing related documents. The answers obtained from each layer are then combined together to form the final answer to the query.

As an application scenario, consider a community (e.g., an organization or a university). The users and their computers can be viewed as individual peers who share and exchange their documents. The documents may have been downloaded in the process of a Web search, or authored by the users. A user can pose a *query* (e.g., "I am interested in papers on information retrieval") and the system returns a list of matching resources.

2 Multi-Layer Graph Model (MLGM)

The Multi-Layer Graph Model (MLGM) (Figure 1) consists of K superimposed layers. Each layer is organized as a p2p network. Each layer L_i, $i \in [1, K]$, is a directed graph $G = (V, E)$, where V is a finite set of vertices and E is a finite set of edges. Each element in V corresponds to a peer p_i, where $i \in [1, N]$ and N is the total number of peers. Each edge in E is a relation of the form $\langle p_i, p_j \rangle$, denoting a link from p_i to p_j (i.e., E corresponds to the links stored in the routing indices of the peers).

The proposed model suggests a clustering of data entities at each layer so that similar entities are grouped together. Peers may belong to more than one cluster and are logically connected (within each cluster) by virtue of containing similar entities. For routing purposes, each peer maintains a routing index (RI) with two kinds of links to other peers:

Long-range links corresponding to *inter-cluster* information (i.e., links to peers belonging to different clusters).

Short-range links corresponding to *intra-cluster* information (i.e., links to peers with similar interests).

In the following, it is assumed that the peers make documents accessible to other peers in the network. Typically, each document may be represented by a term vector. A term is usually defined as a stemmed non stop-word. Typically, the weight w_{it} of a term t in document d_i is computed as $w_{it} = tf_{it} \cdot idf_t$, where tf_{it} is the frequency of term t in document d_i and idf_t is the inverse frequency of t in the whole document collection. The formula is slightly modified for queries to give more emphasis on query terms. The similarity between two documents d_i and d_j (e.g., a query and a document) is computed according to the Vector Space Model (VSM) [3] as the cosine of the inner product between their term vectors as $Sim(d_i, d_j) = \frac{\sum_t w_{it} w_{jt}}{\sqrt{\sum_t w_{it}^2}\sqrt{\sum_t w_{jt}^2}}$, where w_{it} and w_{jt} are the weights in the two vector representations. Given a query, all documents are ranked according to their similarity with the query.

The proposed approach suggests applying clustering to the set of documents at each layer. The purpose of clustering is to partition the space of documents at each layer into disjoint sets of similar documents. Different aspects of information (e.g., schema information, user or peer profiles) can also be represented by vectors and clustered in separate layers. Then, for a given query we can identify the most promising cluster and in each layer route the query only to peers potentially containing relevant resources.

3 P2P Clustering

In a p2p information retrieval network, where peers contribute large volumes of data, it is rather prohibitive to transfer documents to a central clustering service. Therefore, a distributed clustering algorithm is needed. At each layer, each peer organizes its resources (documents) into groups by clustering. Clustering by bisecting k-means [5] has been proven to be particularly effective (i.e., better than the standard k-means and at least as good as hierarchical clustering methods). Each cluster is represented by the centroid (i.e., the mean vector) of the documents it contains. A new document contributed by a peer is assigned to the most similar group by comparing its document vector representation with all its clusters (in fact with their centroids). The peer may recalculate the groups of data items it contributes to the community (and their corresponding centroids), when a certain number of resources is added, deleted or changed. The centroids computed by each peer constitute a *description* of what the peer contains. The peers are further organized into groups of peers with similar interests using their descriptions and the idea of "walks" [4], a procedure by which peer descriptions are forwarded in the network according to some routing strategy. Each peer obtains information about a set of other peers (i.e., their descriptions and network addresses) with similar interests.

Layer_Search(q, L_i)

input
q : the query
L_i : the i-th layer of the layered graph model
$\mathcal{S}$: the similarity function
RI : $\{\langle p_j, e_j\rangle | p_j$ a peer, e_j the description of the peer, $j \in [1, N]\}$

output
$P = \{\langle p_j, \mathsf{S}_j\rangle | p_j$ a peer, S_j the similarity between q and p_j, $j \in [1, N]\}$

begin
for each entry $\langle p_j, e_j\rangle$ in RI
compute in P the similarity S_j between q and p_j
order P by decreasing similarity S_j
end

Fig. 2: Searching within a single layer.

4 Multi-Layer Search

Query processing is performed in steps. First, the query is matched against peer descriptions at each single layer. This process results in a ranked list of peers, where the peers are ordered by decreasing similarity to the query. Figure 2 illustrates this process. This process indicates the most promising peer clusters to search at each layer. Then, the query is routed to the best matching cluster(s) in the layer and returns a list $r^{\{L_i\}}$ with results of the form $\langle d, \mathsf{s}\rangle$, where d a pointer to a data item and s the similarity between q and d.

The answers $r^{\{L_i\}}$ obtained form each layer are combined into a single answer set ordered by decreasing overall similarity to the query. Since the K layers deal with different information, we need a fair strategy to merge the results. Figure 3 illustrates a heuristic approach for searching multiple layers and for combining their results. More elaborate merging techniques can also be applied [7]. For each candidate answer d_j, its overall similarity to the query is computed as $R(q, d_j) = \sum_{i=1}^{K} w_i \mathsf{s}_{ji}$, where s_{ji} is the similarity of data item d_j with q at layer L_i. The weights w_i reflect the relative importance of the layers according to the query. Appropriate weights can be learned by training. The candidate answers are ranked according to their overall similarity to the query and returned to the user.

5 Conclusion

Structured overlay networks are being introduced as tools for better data organization and more effective query processing in p2p systems. Building-upon the same idea, we propose a multi-layer network model for better organization of the contents for each layer. Different aspects of information (e.g., content, schema information) indicate different categorizations of data in separate layers. The benefits of the proposed approach include better organization of network

Search(q)

input
q : the query
$w^{\{L_i\}}$: the weight (relative importance) of layer L_i, $i \in [1, K]$

output
$R = \{\langle d_j, R(q, d_j)\rangle | d_j$ a data item, $R(q, d_j)$ the similarity between q and $d_j\}$

begin
for each layer L_i
Layer_Search(q, L_i)
get lists $r^{\{L_i\}} = \{d, \mathsf{s}\}$ from each layer L_i
put in set C all candidate data items emerging from lists $r^{\{L_i\}}$ for all L_i
for each candidate data item d_j in C
$R(q, d_j) = \sum_{i=1}^{K} w_i \mathsf{s}_{ji}$
order R by decreasing similarity $R(q, d_j)$
end

Fig. 3: Multi-layer search algorithm.

resources by identification of the concepts (as layers) characterizing the network, as well as easy access to information by searching each layer independently and by combining their results.

References

[1] Arturo Crespo and Hector Garcia-Molina. Semantic Overlay Networks for P2P Systems. Technical Report, Stanford University, May 2003.

[2] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Laboratories, Palo Alto, March 2002.

[3] Gerard Salton. *Automatic Text Processing: The Transformation Analysis and Retrieval of Information by Computer*. Addison-Wesley, 1989.

[4] Christoph Schmitz. Self-Organization of a Small World by Topic. In *Proceedings of 1st International Workshop on Peer-to-Peer Knowledge Management*, Boston, MA, August 2004.

[5] M. Steinbach, G. Karypis, and V. Kumar. A Comparison of Document Clustering Techniques. In *KDD Workshop on Text Mining*, 2000.

[6] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.

[7] Theodora Tsikrika and Mounia Lalmas. Merging Techniques for Performing Data Fusion on the Web. In *Proceedings of 10th Conference on Information and Knowledge Management*, pages 127–134, Atlanta, Georgia, USA, 5-10 November 2001. ACM.

Open challenges in ubiquitous and net-centric computing middleware

Thierry Bodhuin, Gerardo Canfora, Rosa Preziosi, Maria Tortorella

RCOST – Research Centre On Software Technology
Department of Engineering, University of Sannio
Via Traiano, Palazzo ex-Poste – 82100, Benevento, Italy
(bodhuin/canfora/preziosi/tortorella)@unisannio.it

Abstract

In a near future, many hardware devices will be interconnected in large and highly dynamic distributed systems, using standard communication protocols on standard physical links.
Nowadays, such a kind of systems exists only for computers interconnected in TCP/IP networks, or for hardware devices interconnected in small areas by using protocols that are specific to the physical link. Java platform, OSGi and Jini have the capabilities to become the foundation of such middleware.
The widespread interest in the ubiquitous or pervasive systems will give new impulses to Net-Centric Computing (NCC). The paper discusses this aspect and proposes an extension of the cited technologies with reference to the context of living environments. In particular, it proposes the Devices Virtualization layer. It is an abstraction level based on the OSGi solution providing a uniform way to access and control different devices.

1 Introduction

The way in which computers are used is changing respect to the approach proposed by mainframe and client-server systems, where one or more persons, occupying a certain desktop position, use a certain type of computers or terminals. Nowadays, ubiquitous or pervasive systems are proliferating and a computer can be used, through the Internet, by one person not considering his site, at any time and regardless the limitations of the used technology. Practically, a revolution in the field of computation, communication and interaction is taking place, and Weiser's vision described in 1991 is becoming a reality [9].
People's thirst for technological products is growing and, in the next years, more and more computational devices surrounding us will invade our daily life with the aim of improving its quality.
Therefore, objects to be controlled and data to be measured exist all throughout the living environments. For example, in a house there are many electronic devices and lighting, heating and water appliances to be controlled. In order to automatically control these devices and appliances, it is necessary to collect data, such as temperature, humidity, noise, scents, etc., from the environment. This need may regard also office and factory setting [7].

Name	Note
Batibus	Widely used in France. http://www.batibus.com/
Bluetooth	In an expansion stage, but broadly supporting short-range wireless technology. https://www.bluetooth.org/
EIB	(European Installation Bus) with home electronic system (HES) user interface: number 1 in Europe. http://www.eiba.com/
Ethernet, TCP/IP	Widely used in the field of IT.
i-Link	(fire wire, IEEE 1394). Multimedia bus systems. http://www.ieee.org/
LON	(Local Operating Network) from Echelon: widely extended in America. During the last years it is more and more accepted in Europe. http://www.echelon.com/
X-10	Primarily used in the USA http://www.x10.org/aboutx10.html

Table 1. Main communication protocols.

Objects to be controlled are technological products, such as common computers, but also sensors, actuators and smaller and more personal devices, which have networking capability and can be linked by wired and wireless heterogeneous and spontaneous networks. They are provided by different makers and/or can be based on different communication protocols, some of which are listed in Table 1 and/or different service and discovery–focused standards, as shown in Table 2.
This scenario lets to glimpse a reality where ubiquitous or pervasive systems will take place in every kind of living environment (or ambient). For example, a living environment can be professional (offices, shops, laboratories, museums, classrooms), personal (homes, gardens), transit (streets, squares, airports), transport (buses, trains, cars). Authors refer living environment equipped with ubiquitous or pervasive systems as smart living environments or smart ambients.
The cited vision suggests that the world will consist of ubiquitous systems rather than "*the ubiquitous system*" [5]. These systems will have to be conceived like large and highly dynamic distributed systems, always working

and available, such that "browsing reality" will be similar to browsing the web. Today's Internet users browse the Web, tomorrow they will browse reality, exploring their living environments [2].

Name	Note
HAVi	Home Audio/Video interoperability specification. It defines a set of protocols/APIs that include device abstraction /device control models, an addressing scheme/lookup service, an open execution environment, Plug-n-Play capability (through 1394), and management of isochronous data streams. http://www.havi.org/
Jini	It is a middleware layer that resides on top of a Java Virtual Machine and uses Java' s Remote Method Invocation (RMI) to make use of a remote device's services. Jini technology has a lookup service with which devices and services register. http://www.jini.org/
OSGi	(Open Service Gateway initiative). It is platform and communication medium independent and supports multiple local network technologies whether wired or wireless. It also supports device access technologies as UPnP and Jini as well as java as an implementation platform. http://www.osgi.org/
UPnP	(Universal Plug and Play). It is an initiative for the discovery and service APIs for a dynamic environment as might be found in the home. http://www.upnp.org/

Table 2. Main service and discovery-focused standards

Authors believe that the widespread interest in the ubiquitous or pervasive systems will give new developments to Net-Centric Computing (NCC). In particular, it will influence the notion of *thin client* and, consequently, that of NCC middleware.

In reality, a wider *thin client* definition should be explicitly considered for including networked devices, smart sensors, actuators that can be linked to broadband, heterogeneous, wired and wireless networks, and other typical technological products oriented to smart living environments. Authors refer this class of thin clients as *ubiquitous thin clients*. As a consequence, new pieces of middleware should be proposed in order to take into account a wider definition of *thin client*. In addition, researchers' attention should be focused on the possibility of integrating OSGi (Open Service Gateway initiative) framework [6] with NCC existent middleware.

The OSGi framework represents a common environment hosting bundles, which are Java archives. It manages the life cycle of the bundles and solves their interdependence, searches classes and resources the bundles make available, keeps a registry of services and manages the events informing the listeners when the state of a bundle is changed, a service is stored or an error occurs.

Besides OSGi solutions, other discovery-focused standard exist. They are complementary, rather than competitive, even if they are sometimes partially overlapped in some provided facilities. However, OSGi appears to be the most natural extension of existent technological solution for NCC.

The following section considers some background regarding technologies to be used; the subsequent section describes the Devices Virtualization layer; final remarks and some open questions will be given in the last section.

2 Background

Java, Jini and CORBA are traditional solutions for NCC [3, 8]. Together they offer a complete distributed object framework in support of net-centric applications. Nowadays, Java and Jini are the preferred software middleware for NCC. In particular, the power of Java comes from its portability as code is written once and runs anywhere [3]. Jini is a Java specific middleware, based on Java RMI [4] in support to NCC. It is designed for creating and accessing services, and allowing anything with a processor, memory, and a network connection, to offer services to other entities on the network or to use the services the other entities offer [8].

In the last year, combination of Java and Jini allow to build ubiquitous distributed infrastructures and these classes of entities can include not just traditional thin client, but also PDAs, mobile phones, televisions, ovens, and so on. Unfortunately, this solution does not permit the life cycle management of services through a services' container.

On the other hand, OSGi has all the requirements for enabling the life cycle management of services. One particular application domain of OSGi uses it as residential gateway that connects devices to the Internet. In fact, the use of OSGi as a gateway permits the connectivity between different internal networks (e.g., X-10, CAN bus, EIB, …) letting hardware devices to interoperate and external networks (e.g., Internet) for controlling and monitoring the devices through different user interfaces (e.g.: Web/HTML, WAP/WML, Java Swing, …). OSGi defines a Java-centric set of APIs to allow devices to be bound to the residential gateway and, thereby, to export their services for being remotely accessible. OSGi can be adopted by a broad category of vendors of heterogeneous devices. Moreover, OSGI and Jini technologies may be integrated for providing a complete distributed services management framework preserving existent Jini/NCC solutions.

Finally, in order to free the developers from the burden of implementing low level interaction with the ubiquitous thin client and provides an event-based communication mechanism for facilitating the interaction between the ubiquitous thin client, the authors suggest to use a further abstraction level named *Devices Virtualization* layer, as shown in Fig.1.

Such a kind of layer has been developed within an on-going research project of the Regional Centre of Competence in Information and Communication

Technology, CRdC ICT. The activities are executed in the Campania Region in Italy. The project involves many research and industrial partners of the Campania Region in Italy. It aims at analyzing, defining and realizing hardware and software platforms for permitting the provision of networked services and the implementation of advanced technologies. In particular, the activities carried on in the operative unit of the University of Sannio, RCOST (Research Centre On Software Technologies), aim at developing a platform in the field of home automation [1]. The platform addresses the following:

- Virtualization of devices, for making independent the applications from the characteristics (hardware, protocol, drivers, etc.) of a particular device.
- Description of devices, for defining a functional and semantic characterization of the devices.
- Description of services, for providing a functional and semantic characterization of the services with reference to their relations with the other services and devices.

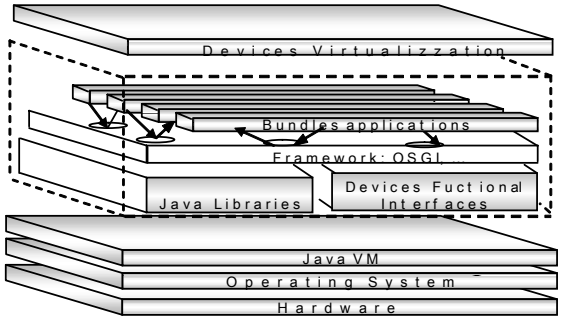


Figure 1. Architecture for traditional and ubiquitous highly dynamic thin clients

3 Devices Virualization layer

Fig. 1 shows the proposed software architecture for traditional and ubiquitous highly dynamic thin clients. The combination of Jini with OSGi permits to extend the thin client definition and to preserve existent NCC solutions by performing a little integration effort with the OSGi Framework.

The *Devices Virtualization layer* is the upper layer in the figure. The functional objective of this layer is to provide an abstraction of the ubiquitous thin clients, installed in the living environment and connected to the hardware layer. This layer regards the functional characterization of the ubiquitous thin clients and contains the implementation of the software objects modelling them. The interfaces of these objects are maintained in the component *Devices Functional Interfaces* of the below layer. The thin client generalization, defined on the basis of the working modalities of the considered devices, has been made independently from their type, nature, semantics and used communication protocol.

The lowest layers in Fig.1 form the system infrastructure and include the *Operating System* and the *Java Virtual Machine* (*VM*) and *Hardware* and needed drivers.

The realization of the Devices Virtualization layer requires the execution of the activities of analysis, modelling, and consolidation.

The *analysis* phase allows observing that changing the state of an area of a living environment substantially means triggering actions on the present devices for producing effects. The effects may be measured and a chain of events may or may not be triggered in the Devices Virtualization layer, involving other changing state in the same area or other environments. It is possible to define a functional view classifying the devices in two main families:

a) *Sensors*, capturing information from the networked devices and/or the environments, and, then, producing events;
b) *Actuators*, consuming events and, then, triggering actions on the networked devices in the considered environments.

Two different devices have different identity (type) expressed from a set of attributes, regarding *name*, *serial*, *version*, *model*, *manufacturer*, etc. Two devices with different nature are logically connected to two distinct physical concepts. Nevertheless, two devices with distinct type and nature may share the same actuation mechanism. For example, a networked lamp is a device different from an alarm. The lamp is logically connected to the electric light concept and may change the state of the environment area where it is installed by providing or not providing light on the basis of the switch on/off actuation mechanism. Similarly, the alarm is logically connected to the sound concept and may change the state of the environment area hosting it by providing or not providing noise in accordance with its open/close actuation mechanism. Therefore, the lamp and alarm are devices of different type, nature and semantic but they share an actuation mechanism with the same working procedure.

Sensors and Actuators can be still specialized in other objects. For example, the networked rolling shutter has a mechanism of actuation different from that of the networked lamp and alarm. In fact, it cannot be defined on the basis of two values but by considering a set of valid values. For example, the rolling shutter may have five possible valid values, *absent*, *low*, *medium*, *high*, *highest*, modelling five different positions and brightness degrees.

Besides the Sensors and Actuators, complex devices exist in the living environments. They are the result of the composition of more elementary devices. For example, a camera is defined as a complex device with different

elementary actuation mechanisms related to different functionalities, as later described.

The *modelling* phase aims at implementing the functional behaviour of each functional object identified in the analysis phase.
Fig.2 shows a simplified view of the designed device interface hierarchy. This figure evicts that:

a) It is possible a first level of specialization of the generic devices of type Sensor and Actuator. For example, the networked lamp is a device of Actuator type which can be characterized by a *BinaryActuator* interface being able to assume only two valid values. While the rolling shutter is a device of Actuator type, which can be characterized by a SetValueActuator, interface assuming different discrete defined values. Furthermore, a device assuming values inside a given continuous range can be characterized by a *RangeValuesActuator* interface. Besides those discussed, further specialization levels can be identified.

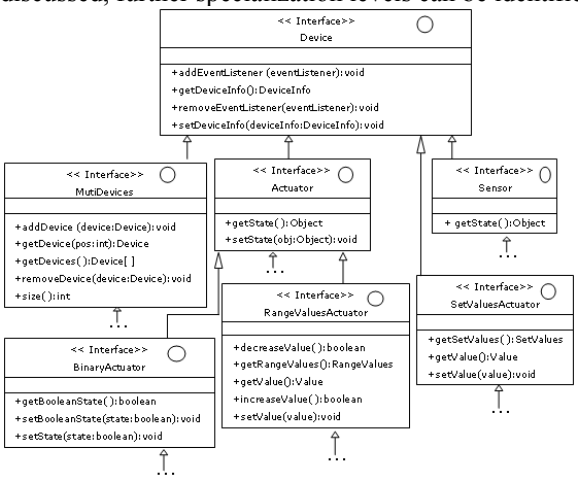


Figure 2. A simplified view of device interface hierarchy

b) Interface *Device* is characterized by methods adding and removing the *EventListener* objects. These methods are used from clients for registering and un-registering listener in *Device*. Thus, clients can be notified, in a push way, of the changes in the state of the devices for taking their decisions. Listener and event interface hierarchies are also defined.

c) Interface *Device* is characterized by getting and setting methods for accessing and/or manipulating the *identity* of a considered device; its handling is a first step toward the modelling of devices that consider the semantic aspect.

d) Complex devices, such as a camera, can be considered a specialization of the *MultiDevices* object. Figure 3 shows the composite pattern used for developing this kind of object. This solution offers the possibility to model any complex device in a simple and extensible way. In fact, Figure 3 exhibits an interface declaring methods for adding/removing single devices to/from the set of devices composing the considered complex device.

The interface hierarchy shown in Figure 2 is not complete. It is shown that it permits the realization of reusable software components and that the Devices Virtualization layer is still valid even when the hierarchy is extended for including new devices, independently from their complexity.

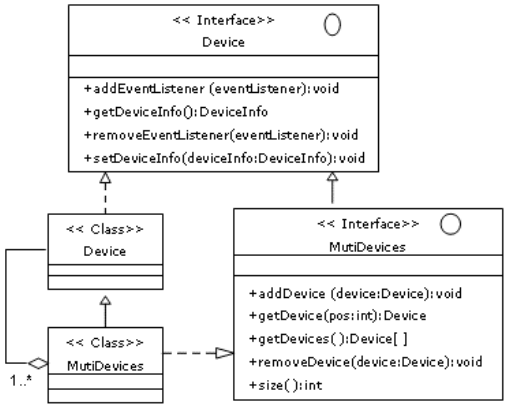


Figure 3. Interface MultiDevices

During the *consolidation* phase, the designer has the opportunity to clean up the designed model, improve its structure and comprehensibility, and, then, increase its potential reuse.
The Devices Virtualization layer has been implemented so that it can evolve. Actually, the devices in the living environment, the inhabitants and their behaviours, may evolve in time and the Devices Virtualization layer has to continually adapt itself to the changing exigencies. Bearing this in mind, the consolidation phase has to be characterized by the execution of evolutionary prototyping and refactoring processes.
Evolutionary prototyping is suitable for developing applications whose the requirements are not fully known and/or continuously changing. It is a good technique for decreasing uncertainties and misunderstandings with reference to the organization of living environments and the functionalities provided by the installed networked devices. In addition, prototyping permits to constantly measure the usability of the prototypal architectural design and reuse the most efficient modules.
Refactoring was applied for moving attributes and methods between classes and code in the hierarchies of classes and interfaces. For example, it involved the listener and event hierarchies that were designed for supporting the events of state change and their propagation to the registered listeners. At the first stage, the listener and event hierarchies had the same structure of the devices interface hierarchy. In particultar, the listener hierarchy contained a listener for each type of

device. A generalization was, then, realized by using the classes *PropertyChangeEvent* and *PropertyChangeListener*, defined in the JavaBeans APIs. An exception was done for defining the events and listeners concerning localization devices for localizing identities, such as Radio Frequency Identification (RFID). Localization concerned the monitoring of people and/or tags in living environments. The change of the state of a localization device meant a detection of people and/or tags. The exception of this generalization was required for improving the efficiency of the localization mechanism by reducing the granularity of the change event.

4 Conclusions and open questions

The way in which computers are used is changing respect to the approach proposed in the past by mainframe and client-server systems. In addition, other hardware devices different from computers will be directly accessible, through the Internet.

In this context, authors believe that the widespread interest in the ubiquitous or pervasive systems should give new developments to NCC . In particular, it should influence on the notion of *thin client* and, consequently, on that of NCC middleware.

In particular, a wider *thin client* definition should be explicitly considered for including networked devices, smart sensors, actuators that can be linked to broadband, heterogeneous, wired and wireless networks, and other typical technological products oriented to smart living environments. As a consequence, new pieces of middleware should be proposed in order to take into account a wider definition of *thin client*.

With this in mind, the paper proposes an abstraction level named *Devices Virtualization* layer, providing a uniform way to access and control different devices and raising the developers from the burden of implementing low level interaction with the ubiquitous thin clients.

In any case, many aspects could be taken in consideration toward and extension of the NCC concepts. To fulfill this purpose, the following open questions may be answered:

- should the thin-client and NCC middleware concepts to change for integrating ubiquitous computing?
- how should the existent technologies change for supporting the evolution of the thin-client and NCC middleware concepts ?
- how should traditional interface evolve for allowing common people to browse the reality in the most natural way?
- which instruments should NCC developers offer to common people for preserving their security and privacy?
- what may be needed for reaching standardization on virtualization of remotely accessible highly dynamic thin clients?

References

[1] Bodhuin T., Canfora G,. Preziosi R., and Tortorella M., “An Extensible Ubiquitous Architecture for Networked Devices in Smart Living Environments”, Proc. 2nd International Symposium on Ubiquitous Intelligence and Smart Worlds 2005, to appear.

[2] Boone G., “Reality Mining: Browsing Reality with Sensor Networks”, online Magazine Intelligent System, September 2004.

[3] Hamilton M.A., “Java and the Shift to Net-Centric Computing*”*”, IEEE Computer, August 1996, 28(8) pp. 31-39.

[4] JavaSoft, Java Remote Method Invocation, http://www.javasoft.com/products/jdk/rmi/index.html.

[5] Kindeberg T. and Fox A., “System Software for Ubiquitous Computing”, IEEE Pervasive Computing, vol. 1, no. 1, Jan./Feb. 2002, pp. 70–81.

[6] Open Service Gateway Initiative, *The Open Service Gateway*, http://www.osgi.org

[7] Takada T., Kurihara S., Hirotsu T. and Sugawara T., “Proximity Mining: Finding Proximity using Sensor Data History”, Proceedings of the Fifth IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2003).

[8] Waldo, J.,: "The Jini Architecture for Network-Centric Computing", Communications of the ACM, Vol. 42 No. 7, July, 1999.

[9] Weiser, M.: “The computer for the 21st Century”, *Scientific American,* Vol. 265, No. 3, 1991, pp. 94-104.

Position paper for 4th International Workshop on Net-Centric Computing
(NCC 2005: September 25, 2005; Budapest, Hungary)

Building Secure Middleware Using Patterns

Eduardo Fernandez
Computer Science & Engineering
Florida Atlantic University
ed@cse.fau.edu

Shihong Huang
Computer Science & Engineering
Florida Atlantic University
shihong@cse.fau.edu

Abstract

There are numerous applications that require the use of a variety of constantly changing policies. These include medical systems, financial applications, legal applications, and others. The applications in these systems are usually integrated using a Web Application Server (WAS), a type of middleware that has a global enterprise model, typically implemented using object-oriented components such as J2EE or .NET. Middleware is one of the key components of Net-Centric Computing (NCC) technology. These applications are of fundamental value to enterprises and their security is extremely important. A systematic approach is required to build these applications so they can reach the appropriate level of security. This paper describes an approach that addresses security throughout the application lifecycle using security patterns. Patterns provide solutions to recurrent problems and many of them have been catalogued. We see the use of patterns as a fundamental way to incorporate security principles in the design process even by people having little experience with security practices.

Keywords: security, middleware, patterns

Position paper to the *4th International Workshop on Net-Centric Computing*
(NCC 2005: September 25, 2005; Budapest, Hungary)

Identifying Security Concerns of Middleware in Net-Centric Computing Environments via Use-Case Analysis

Shihong Huang
Computer Science & Engineering
Florida Atlantic University
shihong@cse.fau.edu

Eduardo Fernandez
Computer Science & Engineering
Florida Atlantic University
ed@cse.fau.edu

Scott Tilley
Dept. of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Abstract

Net-centric computing (NCC) provides a great environment for resource sharing and problem solving among peers across a network. While providing the benefits to users and applications accessing resources that could be distributed in a seamless manner, the architecture of NCC technology exposes a great degree of security risks and threats to the legitimate use of an NCC operational environment. This paper identifies some of policies that could prevent the security threats and risks to the middleware through use-case analysis. We adopt the UML 2.0 standard and the use of Interaction Overview Diagrams and annotated Activity Diagrams to describe security policies that can prevent threats and risks of middleware, and further reduce the security vulnerability of NCC architecture.

Keywords: security, UML 2.0, use-case, middleware, net-centric computing

Towards Developing Grid Middleware for Software Evolution

Jianzhi Li and Hongji Yang
Software Technology Research Laboratory,
De Montfort University, Leicester, LE1 9BH, England
{jianzhli, hyang@dmu.ac.uk}

Abstract

Grid middleware simplify Grid programming by raising the level of abstraction and reducing the accidental complexities. This paper tries to integrate middleware technology and Grid technology for legacy software evolution. This work aims at the design and development of a Grid Middleware for "Software Evolution Grid". In particular, we research to develop Grid middleware foundations for legacy applications evolution, which will provide transparent access to legacy resources concerning process, achieve continuous computation and make better utilisation of the available computing resources among distributed environment.

Keywords: Grid, Middleware, Software Evolution, Legacy System, distributed system.

1. Introduction

Net-Centric Computing (NCC) is a distributed environment where applications and data are downloaded from servers and exchanged with peers across a network on as as-needed basis. It attracts significant interest from network and telecommunications practitioners as a vehicle for innovation. As an emerging trend in Net-Centric Computing technology, Grid computing [4] enables users to collaborate securely by sharing processing, applications, and data across systems to facilitate collaboration, faster application execution, and easier access to data. Grid focuses on large-scale resource sharing, innovative applications and high performance orientation. The sharing relationships may be static and long-lived or highly dynamic.

With increasing adoption of distributed systems and Web based applications, more and more existing applications turn to legacy systems. Grid technology can be applied to achieve enterprise-wide applications integration. Adapting Grid to evolving legacy systems could provide a rich set of capabilities. It allows organisations to use numerous computers to solve problems by sharing computing resources [8].

As these organisations vary tremendously in their purpose, scope, size, duration, and structure; particularly, the resource configurations of organisations have the potential to change dramatically. Providing Grid middleware for deploying existing applications is very useful to enable dynamic load distribution, fault resilience, ease system administration and data access locality.

2. Integration of Middleware and Grid

With years of efforts, Grid researchers have successfully developed Grid technologies including security solutions, resource management protocols, information query protocols, and data management services. However, as the goal of evolving legacy software into Grid computing environment, it is necessary to design an infrastructure to support dynamic, cross-organisational resource sharing, which acts as a solution for efficient and transparent task re-scheduling and management in the Grid.

Middleware is reusable software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware. Appling middleware technology in Grid network is an attractive way for re-scheduling tasks in distributed environment.

Grid middleware simplify Grid programming by raising the level of abstraction and reducing the accidental complexities. With the support of process coordination [2], various runtime load balancing schemes can be employed for improving the execution efficiency of coarse-grained Grid applications. It helps those long-running applications by relocating them at suitable times to prevent interruption due to system activities or the execution of other applications [3]. It also can help relocate processes closer to the Grid point with data that they need to access.

Legion [6] was the first integrated Grid middleware architected from first principles to address the complexity of Grid environments. Just as a traditional operating system provides an abstract

interface to the underlying physical resources of a machine, Legion was designed to provide a powerful virtual machine interface layered over the distributed, heterogeneous, autonomous, and fault-prone physical and logical resources that constitute a Grid.

[5] proposed an ecological network-based Grid middleware (ENGM), which is based on ecological network computing environment (ENCE). It provides a new computing and problem-solving paradigm by combining natural ecosystem mechanisms with agent technologies and support desirable requirements of new Grid systems, namely scalability, adaptability, self-organisation, simplicity, and survivability.

Scalable Inter-Grid Network Adaptation Layers (Signal) [7] is a middleware architecture which integrates mobile devices with existing Grid platforms to conduct peer-to-peer operations through proxy-based systems.

The Grid Application Toolkit (GAT) [1] provides a unified simple programming interface to the Grid infrastructure, tailored to the needs of Grid application programmers and users. Its implementation handles both the complexity and the variety of existing Grid middleware services via so-called adaptors.

To facilitate transparent use of the high-performance Across Trophic-Level System Simulation (ATLSS) ecosystem-modeling package for natural-resource management, [10] developed a Grid service module. The module exploits Grid middleware functionality to process complex computation without requiring users to handle underlying issues.

NAREGI [9] aims to research and develop high-performance, scalable Grid middleware for the national scientific computational infrastructure. Such middleware will help facilitate computing centers within Japan as well as worldwide in constructing a large-scale scientific “Research Grid” for all areas of science and engineering, to construct a “National Research Grid.

3 Discussion

As an interface to applications, Grid middleware may have following performance:

- Track which node has computational servers running and which one are provisioned with.
- Track each node’s workload to locate the best choice for a given job request.
- Take care of the details in finding machines on which to execute computational tasks.

In this study, our work aims to research and develop high-performance, scalable middleware based on Grid technology oriented approach for “Software Evolution Grid”. More specifically, we research to develop Grid middleware foundations for legacy system evolution. Such middleware will provide transparent access to legacy resources concerning process, achieve continuous computation and make better utilisation of the available computing resources among these organisations. In addition, we investigate the architecture of large, distributed, computing infrastructures, high-speed networks to support legacy systems, and methodologies for effective software evolution on the underlying distributed infrastructure.

A Grid provides an abstraction for resource sharing and collaboration across multiple administrative domains. Resources are physical (hardware), informational (data) and capabilities (software). From the software evolution perspective, legacy resources are encapsulated into objects which can be used by the developers as components for their applications. Grid middleware can be acted as software that facilitates writing these applications and manages the underlying Grid infrastructure.

Comparing with design a new system, our “Software Evolution Grid” approach is less risky and highly transport, and it is massively reduce time and cost.

4. References

[1] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T.Schott, E. Seidel, and B. Ullmer, "The grid application toolkit: toward generic and easy application programming interfaces for the grid," *Proceedings of the IEEE*, v93, n3, March 2005, p 534-50.

[2] X. Bai, H. Yu; G Wang; Y. Ji; M. Marinescu; C. Marinescu and L.Boloin, "Coordination in intelligent grid environments," BoloniSource**:** *Proceedings of the IEEE*, v93, n3, March 2005, p 613-30.

[3] L. Chen, C. Wang anf F Lau, "A Grid Middleware for Distributed Java Computing with MPI Binding and Process Migration Supports," *Journal of Computer Science and Technology)*, v18, n4, July 2003, p 505-14.

[4] I. Foster, C. Kesselman and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organisations," *International Journal of High Performance Computing Applications*, vol. 15, no. 3, 2001, pp. 200-222.

[5] L. Gao, Y. Ding and L. Ren, "A novel ecological network-based computation platform as a grid middleware system," *International Journal of Intelligent Systems*, v19, n10, Oct. 2004, p 859-84.

[6] A. Grimshaw and A. Natrajan, “Legion: lessons learned building a grid operating system," *Proceedings of the IEEE*, v93, n3, March 2005, p 589-603.

[7] J. Hwang and P. Aravamudham, "Middleware services for P2P computing in wireless grid networks," *IEEE Internet Computing*, v8, n4, July-Aug. 2004, p 40-6.

[8] J. Joseph and C. Fellenstein, "Grid Computing," *IBM Press*, ISBN: 0131456601 1st ed., 2004.

[9] S. Matsuoka, S. Shinjo, M.Aoyagi,S. Sekiguchi, H. Usami and K. Miura,."Japanese computational grid research project: NAREGI," *Proceedings of the IEEE*, v93, n3, March 2005, p 522-33.

[10] D. Wang, E.Carr, M. Palmer,M. Berry,and L. Gross, L.J., "A grid service module for natural-resource managers Source," *IEEE Internet Computing*, v9, n1, Jan.-Feb. 2005, p 35-41.

Position paper for 4th International Workshop on Net-Centric Computing
(NCC 2005: September 25, 2005; Budapest, Hungary)

Incorporating Net-Centric Computing into Software Engineering Course Projects

Scott Tilley

Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Abstract

Creating software engineering course projects for undergraduate students is a challenging task. The instructor must carefully balance the sometimes-conflicting goals of academic rigor and industrial relevance. The emergence of net-centric computing (NCC) as a cross-disciplinary activity area offers one possible solution to this problem. Since NCC incorporates many elements of computing, such as software construction, distributed systems, networks, security, databases, and Web sites, it can provide a rich foundation upon which exciting software engineering course projects can be built. The paper presents some early experience in using NCC in this context, and outlines future collaborative projects.

Keywords: software engineering education, course projects, net-centric computing

"Change minder": Towards a general web-change notification system, based on HTML differencing

Eleni Stroulia and Rimon Mikhaeil
Computing Science Department
University of Alberta
Edmonton AB, T6G 2H1, Canada
{stroulia, rimon} @cs.ualberta.ca

Abstract

Today, web users regularly visit several web sites providing information and services in different areas of interest and activity. As the number of their web sites of interest increases, visiting them regularly becomes increasingly time consuming, especially since their content is not always updated at a regular rate. Some sites offer RSS-feeds to alert their subscribers to changes, however most are not. In this paper, we describe an algorithm for differencing HTML documents that can be used to support a general "change-minding service".

1 Introduction

HTML "differencing" is essential to monitoring and recognizing recent updates to web pages of interest that may be frequently and irregularly updated, like a course home page, a news page, etc. As web users become more experienced, their set of "pages of interest" is likely to increase and such an automated "update-recognition service" becomes increasingly relevant.

The comparison of HTML pages can be based simply on the textual content of these pages, using traditional document-comparison approaches. Unfortunately, such approaches suffer from a critical drawback: it misses the HTML's structural features. For example, comparing the contents of a simple page, shown in Figure 1(a), to the one shown in Figure 1(b) using document comparison mechanisms may result in an output similar to the one shown in either Figure 2(a) or Figure 2(b). A document comparison approach would report that the entire content of the first document was deleted while a new content was inserted into the second one. However, both documents have the same content with some structural modifications. Hence, a text-based comparison does not work properly with HTML as it has a structure that should be significantly considered.

This is exactly the basis of the work reported in this paper. We consider an HTML page as a tree structure that may have suffered modifications to either its structure or its content. Hence, we have adapted a standard tree-comparison algorithm to the HTML-comparison problem.

```
<html>
<body>
<b>test text</b>
</body>
</html>
```

(a) page1

```
<html>
<body>
<a href="">test text</a>
</body>
</html>
```

(b) page2

Figure 1 Two pages to be compared

```
1:  <html>
2:  <body>
3:  <a href="">test text</a> </body>
4:  <b>
5:  test text
6:  </b>
7:  </body>
8:  </html>
```

(a) by [HTML Match]

```
<html>                <html>
<body>                <body>
<i>test text</i>      <b>test text</b>
</body>               </body>
</html>               </html>
```

(b) by [Diff Doc]

Figure 2 Comparison Results

2 HTML differencing as tree comparison

By tree comparison, we refer to the process of deciding the minimum number of editing operations (i.e., adaptation, deletion, insertion, and movement) that reasonably transform the first tree to the second one. For example, a comparison between the trees shown in Figure 3 would yield that:

1) element *a* was changed to *a'*,
2) element *d* was moved,
3) element *e* was mapped,
4) element *c* was deleted,
5) element *l* was inserted, and
6) element *b* was mapped.

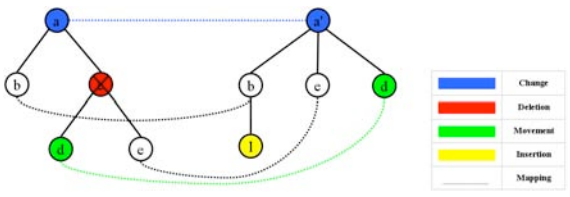
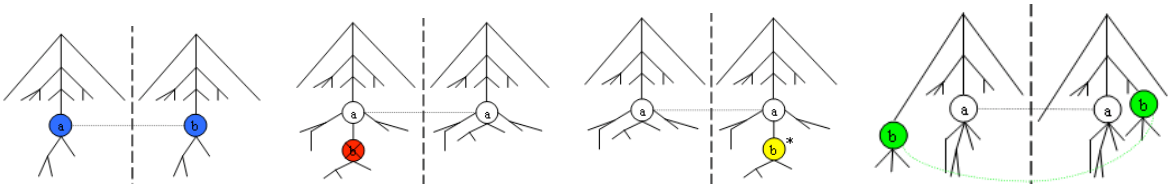


Figure 3 The Desired Tree-Comparison Result

A *mapping* operation is the simplest operation. Actually, it is not an editing operation at all, i.e. it implies that a node has not changed. As shown in Figure 4 (a), a *changing* operation is very similar to a mapping operation but according to the given cost function, it is been mapped to a similar as opposed to the same node. As shown in Figure 4 (b), a *deletion* operation is to map a node (from the first tree) to nothing in the second tree, when a suitable counterpart for that node cannot be found in the second tree. As shown in Figure 4 (c), an *insertion* operation is the inverse of the deletion operation where a counterpart for a node in the second tree does not exist in the first tree. Finally, as shown Figure 4 (d) in a *movement* operation, a certain node has a counterpart in the second tree, although not in the same relative order to its siblings. Typically, this operation may be recognized as deleting that node from the first tree and inserting it elsewhere in the second one.



(a) Changing operation **(b) Deletion operation** **(c) Insertion operation** **(d) Movement operation**

Figure 4 Editing Operations

3 Approach overview

This section presents an overview of the original Zhang-Shasha tree-comparison algorithm on which our own HTML-comparison is based [1, 2]. Then, we discuss our proposed path recovery mechanism. Finally, we introduce a new editing operation called *movement* operation.

3.1 Tree Preparation

The nodes of the two trees to be compared are labeled with their pre-order traversal index. For example, the tree in Figure 5(0) is to be counted as shown in Figure 5(1).

Next, its key sub-trees (shown in Figure 5(2)) are identified as follows. Every leaf node identifies a certain *key* sub-tree dominated by a corresponding *key root*. A key root is the upper-most ancestor of that leaf node that is not defined as a key root of another leaf.

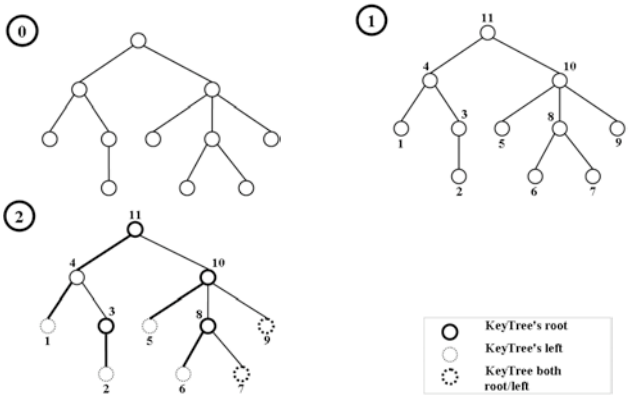
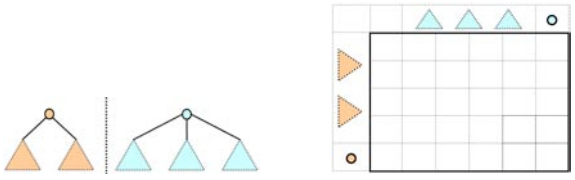


Figure 5 Tree Preparation

3.2 The main algorithm

One implementation of the Zhang-Shasha's algorithm [2] is based on a 2-d dynamic programming matrix where rows represent the post-ordered nodes of first tree while columns are for the post-ordered nodes of the second one. As shown in Figure 6, each cell (*i, j*) in this matrix represents the cost of transforming the sub-tree rooted by the i^{th} node in the first tree to the sub-tree rooted by the j^{th} node in the second tree. In other words, *Cost*[*i, j*] is the cost of transforming $T_1[l(i)..i]$ to $T_2[l(j)..j]$.

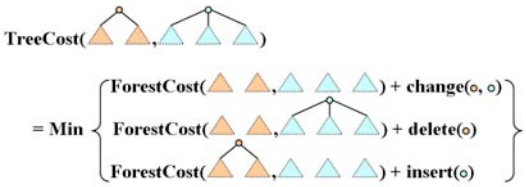


(a) (b)

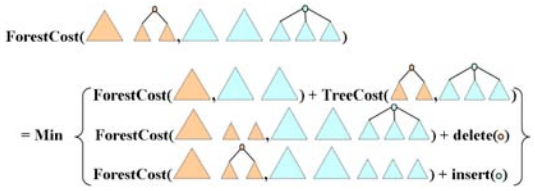
Figure 6 Computational Model

As shown in Figure 7(a), the cost of mapping two trees, $T_1[l(i)...i]$ and $T_2[l(j)...j]$, is the minimum cost among the following three editing options.

1) mapping the root of the first tree *change(i,j)* while trying to map the remaining forests to each other, i.e. to calculate the cost of transforming $T_1[l(i)...i\text{-}1]$ to $T_2[l(j)...j\text{-}1]$;
2) deleting the node *i* and trying to map the remaining forests to each other, i.e. to calculate the cost of transforming $T_1[l(i)...i\text{-}1]$ to $T_2[l(j)...j]$;
3) deleting node *j* and trying to map the remaining parts to each other, i.e. to calculate the cost of transforming $T_1[l(i)...i]$ to $T_2[l(j)...j\text{-}1]$.



(a) Tree-to-Tree Cost [2]



(b) Forest-to-Forest Comparison [2]

Figure 7 Tree Comparison [2]

The *candidacy matrix* (shown in Figure 8) is maintained in parallel to the main cost matrix: for each cell in the main matrix with a calculated cost, the candidacy matrix registers the chosen direction(s) at that cell. Each cell may choose one or more directions out of the available three ones, e.g. up for deletion, left for insertion, or diagonal for mapping/replacing.

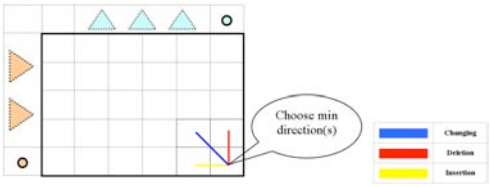
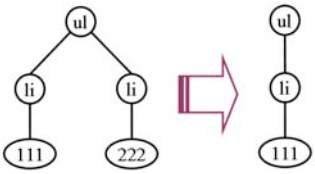


Figure 8 Tree Editing candidacy map

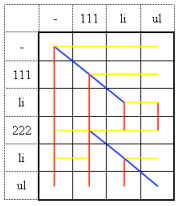
It may happen that at a certain cell more than one direction give the same minimum cost value, so, all of these candidate directions should be registered in the candidacy matrix. Implementation-wise, these cells may be defined as integer holders and each direction is identified by a different number that is a multiple of 2, e.g. up is 1, left is 2 and diagonal is 4. Hence, to specify that a certain cell has more than one candidate direction, all we have to do is to simply apply a bitwise-OR operation between all these directions.

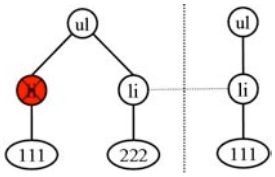
3.3 Affine cost

According to the original algorithm, all the deletion/insertion operations have costs that are independent of the operation's context. For example, a node's deletion cost is the same in case its children are deleted or not. As shown in Figure 9, the former policy would result that the second "li" in the first tree is always mapped to the only available "li" in the second one and it would always delete the first "li" from the first tree, which is not a realistic transformation.



(a) Trees to be compared



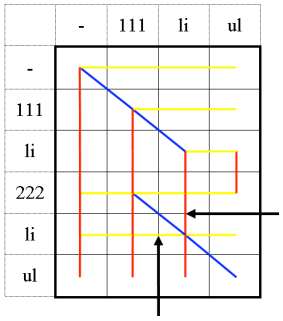


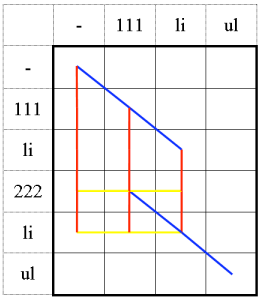
(b) Candidacy map (c) Reported result

Figure 9 Non-affine cost function

To produce more plausible results, we use what is so called *affine cost policy:* a node's deletion/insertion cost is not the same in all cases but depends on its context. If a node's all children are candidates to be deleted, so this node is more probable to be deleted also. The same with a node having all its children sub-trees are candidates to be inserted. To reflect this heuristic, the cost of the deletion/insertion of such a node should be less than usual by just adding a small amount (<actual cost) to the cost of deleting/inserting its children.

For a node i to be eligible for deletion affine cost, the candidacy map should have a straight vertical line $(i\text{-}1,j)$ up to $(l(i),j)$; a straight vertical line means that all the children are candidates for deletion. Similarly, cell(i,j) is eligible for insertion affine cost if the candidacy map has a straight horizontal line begin at $(i,j\text{-}1)$ to $(i,l(j))$. For example, applying this mechanism on the example of Figure 9(a) would result in Figure 10(a) instead of Figure 9(b); where we note that the directions pointed by the couple of arrows do not exit in the former map, i.e. this "li" was eligible for both deletion and insertion affine costs.





(a) Using affine cost function (b) cleaned map

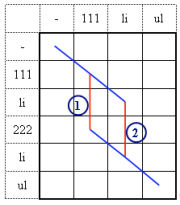
Figure 10 Editing map with affine cost applied

3.4 Editing operation-sequence recovery

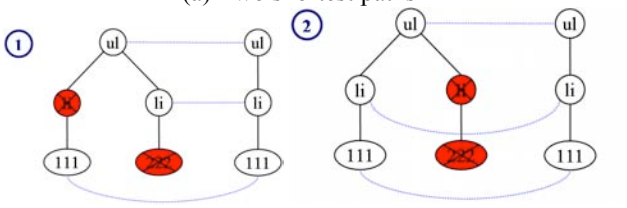
The original Zhang-Shasha algorithm reports the minimum number of editing operations that transforms the first tree into the second one but does not report what these operations are. In this paper, we propose a couple filtering phases, applied on the candidacy matrix, to reach the best path.

3.4.1 Shortest paths

Our path recovery mechanism is based on the heuristic that *shortest path is better*. In this phase, the candidacy matrix is scanned from the lower-right corner up to the upper-left one: for each cell, the shortest path from that cell up to the upper-left corner is calculated, discarding directions that does not lead to that shortest path. Figure 11(a) shows the candidacy matrix of Figure 10(b) after discarding the non-shortest paths.



(a) Two shortest paths



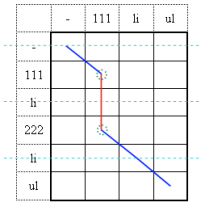
(b) Path1's transformation (c) Path2's transformation

Figure 11 A map with more than one shorest candident paths

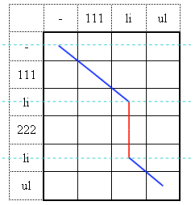
This filtering phase may not result in a single shortest path. For example, the first path of Figure 11(a) would result in the transformation of Figure 11(b), while the second path results in the transformation shown in Figure 11(c). It is obvious that the second transformation is logically preferred over the first one, as it deletes a whole sub-tree. Hence, the next filtering phase is to reflect such a preference.

3.4.2 "Simplest" paths

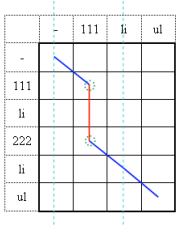
The second filtering phase is based on the idea that between two paths with the same cost, the path that applies homogenous (similar) editing operations on the maximum number of sub-trees is preferable. In other words, the path with the *least number of refraction points* is preferable. By a refraction point, we refer to a node that has an editing operation that is different from its parent's. In order to count the number of refraction points within a certain transformation path, we have to sum the refraction points with respect to both the first and second tree, i.e. number of horizontal refractions plus the number of vertical ones. For example, consider the two shortest paths of Figure 11; namely *path1* and *path2*, then Figure 12 shows that path1 horizontally (with respect to the first tree) has a couple of refraction points while path2 has zero points. Additionally, it shows that path1 vertically (with respect to tree2) also has a couple of refractions while path2 has zero points. Therefore, with respect to the whole transformation, path1 has a total of four refractions while path2 has zero. So, path2 is preferred over path1.



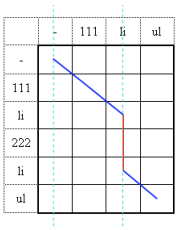
(a) Path1 with respect to T_1



(b) Path2 with respect to T_1



(c) Path1 with respect to T_2

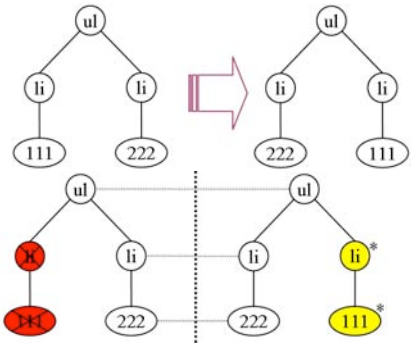


(d) Path2 with respect to T_2

Figure 12 Refraction points investigation of path1 against path2 with respect to both trees

3.4.3 Movement recognition

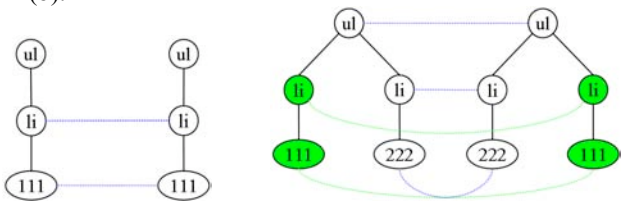
Applying the comparison mechanism explained so far for the trees (shown in Figure 13(a)) would report the result shown in Figure 13(b). The difference between these trees is that a sub-tree (of li and 111) has been just moved to be after another sub-tree (of li and 222). However, the comparison reports that the sub-tree was entirely deleted from the first tree while another (different) sub-tree was inserted into the second tree. Therefore, the issue is that how to recognize such an editing as movement instead of deletion and insertion.



(a) A couple of Trees **(b) Reported editing**

Figure 13 Recognizing movement operations using regular comparsion approaches

Our proposed idea is based on a post-processing phase done to the above reported results. After running the basic algorithm, we try to map whole deleted sub-trees to whole inserted ones. For example, re-applying the above algorithm to the only deleted/inserted sub-trees reported in Figure 13(b) would result that both "li" is mapped to "li" and "111" to "111". Finally, the mapped sub-trees, reported in the second round (shown in Figure 14(a)), reported as movement for the final result shown in Figure 14(b).



(a) Second round **(b) Final result**

Figure 14 Result's post-processing

Specifically, the keyword in this mechanism is to map *whole* deleted sub-trees to *whole* inserted ones. In other words, we don't re-apply the algorithm for all the deleted/inserted nodes. Additionally, we recognize movement operations to deleted sub-trees that are mutually mapped to entire sub-trees. It is worthy to mention that we may repeatedly re-apply the basic algorithm until no new movement operations are recognized.

4 Evaluation

We have developed an HTML-difference visualization system based on this algorithm, which we have compared the usability and intuitiveness of its results against HTML Match (htmlmatch.com) and HTML Diff (aaronsw.com/2002/diff). Our initial data indicate that our tool delivers a more plausible comparison result in most cases, with the exception of pages that have been drastically changed both in terms of their structure and content.

5 References

[1] K. Zhang, R. Stgatman, and D. Shasha. *Simple fast algorithm for the editing distance between trees and related problems*. SIAM Journal on Computing, 18(6):1245--1262, 1989.

[2] Serge Dulucq, and Laurent Tichit. RNA Secondary structure comparison: exact analysis of the Zhang--Shasha tree edit algorithm, Theoretical Computer Science, Volume 306, Issue 1-3 (September 2003): 471 - 484

Position paper for *4th International Workshop on Net-Centric Computing*
(NCC 2005: September 25, 2005; Budapest, Hungary)

The State of Net-Centric Computing in 2005

Scott Tilley
Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Ken Wong
Department of Computing Science
University of Alberta
kenw@cs.ualberta.ca

Abstract

This paper comments on the state of net-centric computing (NCC) in 2005. From the first NCC workshop in 1997 until today, much has changed. For example, broadband penetration has increased dramatically, while personal computing power and data storage capacity have continued to accelerate. However, some things remain the same. For example, the basic computing experience remains essentially application-focused, with network connections supporting the local environment. Three new developments that stand out are the proliferation of malware and the security implications made possible by the networked environment, the surge in peer-to-peer (P2P) communication, and the dramatic increase in the complexity of software engineering in an NCC context.

Keywords: net-centric computing, middleware, software engineering, P2P

Paper for *1st Workshop on Program Redocumentation for Distributed Systems* (PRDS 2005: September 25, 2005; Budapest, Hungary)

On the Challenges of Redocumenting Net-Centric Computing Applications

Scott Tilley
Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Shihong Huang
Computer Science & Engineering
Florida Atlantic University
shihong@cse.fau.edu

Abstract

Program redocumentation is one approach to aiding system understanding in order to support maintenance and evolution tasks. It relies on technologies such as reverse engineering to create additional information about the subject system. This paper presents an overview of some of the emerging research challenges in redocumenting net-centric computing applications. The challenges include in-the-small issues such as dual-core architectures for embedded systems, in-the-large issues such as the use of Web services as a control integration mechanism, and crosscutting issues such as the increasing use of aspects to manage concerns like security.

Keywords: program redocumentation, net-centric computing, research agenda

Workshop 4.1:
Workshop on Design Patterns Theory and Practice

Improving Design Patterns Modularity Using Aspect Orientation

Mario L. Bernardi, Giuseppe A. Di Lucca

dilucca@unisannio.it, marioluca.bernardi@unisannio.it

RCOST -Research Centre on Software Technology, University of Sannio
Palazzo ex Poste, via Traiano, 82100 Benevento, Italy

Abstract

Design Pattern (DP) implementations may suffer of some of the typical problems related to some deficiencies of Object Oriented (OO) languages that may affect the modularity of the system, and thus its comprehensibility, maintainability, and testability.
Aspect Oriented Programming provide patterns' developers with powerful quantification constructs to better handle modularity and composition that can help to overcome some of the OO design tradeoffs and indirections in typical DP implementations.

A set of appropriate metrics to quantitatively evaluate the modularity and the impact of DP adoptions in OO system has to be identified and used to drive the re-development of DPs by aspect-oriented implementations improving modularity.

1. Introduction

Design Patterns implementation may suffer of some of the typical problems related to some deficiencies of OO languages. Indeed, the way a DP is implemented may heavily impact the overall system structure, and in particular it impacts the modularity of the system, thus affecting its comprehensibility, maintainability, testability too.
The invasive nature of pattern implementations and the scattering and tangling of such an implementation with the code of the remaining components of the system (such as the ones related to the application domain entities) may make it hard to distinguish between the patterns instances code and the code of the 'base' system components. In [2, 4, 5] it has been shown how several patterns from GoF catalog [1] introduce crosscutting that OO abstractions are often unable to well modularise.
Aspect Oriented Programming (AOP) provide patterns' developers with powerful quantification constructs to better handle modularity and composition. These constructs can help to overcome some of the OO design tradeoffs and indirections characterising current typical DP implementations. Composition transparency, optionality, and (un)pluggability are example of modularity properties that can be improved by AOP pattern implementation and that have to be considered when assessing the quality of aspect oriented designs [2, 3].
We propose to identify and use a set of appropriate metrics to quantitatively evaluate the modularity and the impact of DP adoptions on system implementation. This evaluation can be used to drive the design of improved DP by aspect-oriented implementations.
Quantitative comparison between aspect-oriented versions of the patterns and their OO counterparts will let us to validate results but also to explore the wider spectrum of design choices available using AOP languages.

2. Motivations

Object Oriented Design Patterns provide the design of generic solutions to recurring problems [1].
The usage of DPs would increase the quality of OO design, but often developers miss the subtleties of the DP consequences [1] by applying/implementing patterns with wrong variation points to add (unneeded) flexibility.
In these cases the usage of such DPs negatively affects the system overall quality.
However, also when the consequences are well taken into account, DPs may introduce in OO systems some problems related to reuse and maintainability that can be hard to solve within the object oriented paradigm.
Usually, these problems are due to the indirection techniques forcing one or more key interfaces to be implemented introducing a greater overhead.
This issue is dealt in [5] where is shown how AOP can help in restructuring GoF patterns to get more effective anticipation of future changes with a lower overhead.
Recent researches have shown that many DPs involve Crosscutting Concerns (CCC). That is mainly due to the poorness of the composition and quantification constructs in OOP languages that do not allow a good modularization of the concerns. Indeed, by using OO DPs, programmers are forced to add classes, interfaces, methods and attributes inside the code of the components (i.e. classes, packages, etc.) derived from the primary decomposition. The introduction of these elements will produce code scattering and tangling by reducing the

quality of some DP attributes such as reusability, traceability, comprehensibility, maintainability.
AOP constructs are able to better modularize DP concerns. In [2, 4, 6] AOP implementations of GoF [2] patterns are provided; they have better values for properties such as locality, (un)pluggability, composability and reusability.
Then a way to improve the modularity of OO DP is re-implement them by AOP techniques.

3. Using AO to improve DP modularity

The modularity of DP implementation can be improved by re-designing DP by AO. The re-design can be driven by a set of appropriate metrics (to be identified) evaluating the crosscutting of DP concerns: DP implementations characterised by bad values of the metrics will highlight the DP to be re-designed.
Metrics based on aspect mining techniques, as well as metrics based on clone analysis [6, 7, 8] can be used to this aim.
Research in this field is not yet mature, and it is related to just some selected types of patterns; in these cases most of the analyzed patterns introduce crosscutting and a high correlation is between superimposed roles [2] and crosscutting.
Thus a deeper investigation and discussion is required about these issues.

We are carrying out some preliminary studies on a suite of selected pattern implementations from different domains (structural decomposition, data access, communication, management and access-control patterns) to get and analyse quantitative data about crosscutting distribution and modularity properties. The DP to analyze were selected from a DP set wider than the GoF catalog.
As a first result we noted that most of the analysed OO pattern implementations introduce CCC at different levels of degree, as well as that AOP-based reengineering is able to improve their quality.
Three main considerations derived from the first results:
1) Patterns become language idiom

In some cases AOP languages can implement directly the patterns with different degree of flexibility. In our experiments we referred to AspectJ language. Interesting comparisons among different AOP language models can be performed.

2) AOP pattern re-implementation produces no benefits (but results in more overhead)

Some pattern implementations are well handled by OO languages thus they gain no actual benefits by AOP constructs. In these cases AOP implementations can be even worse than the OO ones.

3) Patterns involve crosscutting and role superimpositions.

In these cases pattern implementations introduce in OO systems scattered code while AOP implementations presents better modularity.

Of course, also our studies are not yet mature to give satisfactory answers; however they want to provide some quantitative information to solicit a more general discussion about these issues.

References

[1]. Gamma, E., Helm, R., Johnson, R., Vlissides, J., 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley (1995)

[2]. Hannemann, J., Kiczales, G., 'Design Patterns Implementation in Java and AspectJ', Proc. of Object Oriented Programming Systems Languages and Applications 2002 (OOPSLA ’02), Nov 2002, 161-173

[3]. Hachani, O. , Bardou D., 'On Aspect-Oriented Technology and Object-Oriented Design Patterns' , Proc. of European Conference on Object Oriented Programming 2003 (ECOOP 2003

[4]. Monteiro, M.P. , Fernandes J.M. , 'Towards a catalog of Aspect Oriented Refactorings , Proc. of Aspect oriented Software Developmnet 2005 (AOSD ’05)

[5]. Nordberg Martin E., 'Aspect Oriented Indirection - Beyond Object Oriented Design Patterns' , Proc. of Workshop. Beyond De-sign: Patterns (mis)used, Proc. of Object Oriented Programming Systems Languages and Applications 2002, OOPSLA 2002

[6]. Magiel Bruntink, Aspect Mining using Clone Class Metrics , Proc. of Workshop on Aspect Reverse Engineering 2004 (WARE ’04)

[7]. [7] Garcia, A. et al. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In Software Engineering for Multi-Agent Systems II, Springer, Lecture Notes on Computer Science, 2940, January 2004.

[8]. [8] Garcia, A., Silva, V., Chavez, and C., Lucena, C. ‘Engineering Multi-Agent Systems with Aspects and Patterns’. Journal of the Brazilian Computer Society, 1, 8 (July 2002), 57-72.

Semantics of a Pattern System

Ashraf Gaffar and Naouel Moha

GEODES - Group of Open and Distributed
Systems, Experimental Software Engineering

Concordia University, Quebec, Canada
University of Montreal, Quebec, Canada
E-mail: gaffar@cs.concordia.ca, mohanaou@iro.umontreal.ca

Abstract

The wide acceptance of the "Design Patterns" [5] has encouraged experts in other software domains to formulate their experience into pattern format hoping to make it readily reused by developers. We now have numerous pattern collections covering all aspects of software development from analysis to deployment and refactoring. But developers can be overwhelmed by this large number and the lack of coordination and in consistencies among them. These patterns have many similarities and redundancies which may contribute to misunderstanding and wrong reuse. Some research has proposed standards to writing patterns [1] [7] but they were rarely used because each pattern author prefers to use their own creativity [3] which is often a good thing. We propose another approach to address this problem. In each specific software domain, we collect and pre-process existing patterns by defining, detecting and removing some kinds of redundancies between them. The result is a smaller collection of patterns from different sources that have fewer redundancies which reduces confusion and promotes the proper reuse.

Keywords: Design Patterns, Models, Redundancies, Pattern Relationships, Semantics, Similarities.

1. Semantics of a Pattern System

We show some semantics associated with the concept of Generic Pattern Model and some semantics of equivalent/identical relationships between patterns. We originally applied them to interaction design patterns but they can be extended to other types of patterns. They are showing possible benefits in form of tool-support on top and independent of the core pattern information.

1.1. The eXtensible Minimal Triangle, XMT

Data models generally have syntax and semantics. The XML handbook [6] explains that semantics can be further divided into *semantic labeling* of contents, and *abstract interoperable behavior* as demonstrated in figure 1.

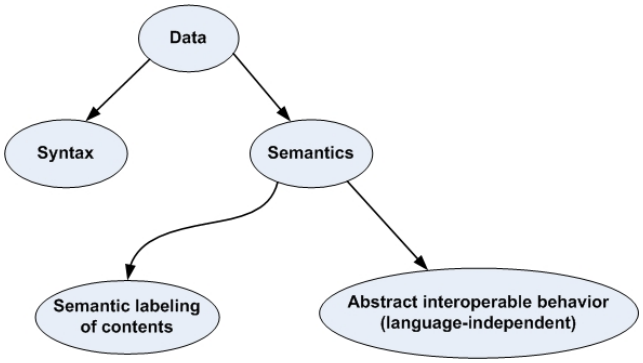


Figure 1. The Data Constituents

In the GPT (Generic Pattern Type) model [2], we defined the syntax of a generic pattern model, and the semantic labeling used for its syntax. Our work here focuses on the abstract interoperable behavior, which involves the behavior underlying the meaning of some of the tags we are using. We start by some preconditions and definitions.

As frequently mentioned in the pattern community, the common denominator of all pattern definitions is "a problem to a solution in a context". Based on that, we defined the minimal triangle as in figure 2.

Definition. *The Minimal Triangle* is a representation of a pattern that has the three elements: a problem, a context, and a solution. No other elements are

present in the minimal triangle. The minimal triangle represents the core meaning of a pattern. Any missing element of the three will result in a trivial pattern.

Definition. *A trivial pattern* is a pattern that has at least one empty element from its minimal triangle.

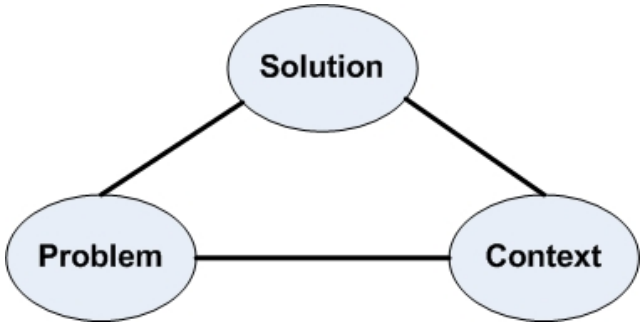


Figure 2. The Minimal Triangle

1.2. Identical Patterns

Informally, identical patterns are the same patterns mentioned in different collections. To be able to use some of our models, we introduce the following formal definitions:

- **Precondition**: Each pattern has a unique ID.
- **Precondition**: Each non-trivial pattern has an existing, non empty minimal triangle (this can be inferred from the definition of trivial pattern).
- **Precondition**: The following definitions strictly apply to non-trivial patterns. For trivial patterns, the definitions can be ambiguous.
- **Precondition**: *Pattern_name(A)* refers to a single object called "the name of pattern A", of type "string".
- **Precondition**: *Pattern_ID(A)* refers to a single object called "the identification of pattern A", of type "string".
- **Precondition**: *Pattern_aliases(A)* refers to the set of zero or more objects called "the aliases of pattern A", of type "set of string objects".
- **Precondition**: There is a one-to-one correspondence between *Pattern_name(A)* and *Pattern_ID(A)*. We emphasize the word "correspondence" to denote a "one-to-one" and an "onto" function between *Pattern_name(A)* and *Pattern_ID(A)*. In other words, the inverse relationship between *Pattern_name(A)* and *Pattern_ID(A)* is also a one-to-one function.
- **Notation (from the set theory)**: (A) $\in$ (B) refers to the object A being an element of the set of objects B.

These are important preconditions that we will use in our next definitions to disambiguate some patterns that are identical or similar at different degrees. Some patterns are already connected to others, but the majority of them are not. We have identified many of these redundancies.

Definition. *Identical Pattern*

Pattern A is identical to pattern B, written A = B, if[1]

Pattern_name (A) = Pattern_name (B) or
Pattern_name (A) $\in$ Aliases (B) or
Pattern_name (B) $\in$ Aliases (A)

1.3. Similar Patterns

As in several other applications, it is sometimes useful to separate between the look (the presentation) and the behavior. The same concept is used in separation of java's look and feel, contents and presentation in CMS (Content Management Systems) and other models. We define the next pattern relationships based on a similar concept: Patterns that look similar and act similar (identical- and similar patterns), and patterns that look different but act similar (equivalent patterns).

- **Precondition**: The next definition is based on the definition of the *Minimal Triangle*.
- **Definition**: The XMT repository specifies three finite sets of keywords corresponding to the three MT items:
  - The first set defines the specific kinds of **problems** covered by the patterns in form of reserved keywords.
  - The second set defines the specific kinds of **contexts** covered by the patterns in form of reserved keywords.
  - The third set defines the kinds of **solutions** covered by the patterns in form of reserved keywords.

[1]This is a sufficient, but not a necessary condition. We can make it necessary by augmenting the aliases set of each pattern with our findings of identical patterns.

- **Definition**: The *Extensible Minimal Triangle* model refers to the three parts *problem, context, and solution* of a given pattern as presented in the GPT and using a subset of reserved keywords elaborated in the *repository* of the XMT model.

The extensibility comes as a key concept of the XMT. If new patterns are to be added, and the keywords of each of its MT parts are not in the space of the XMT, the XMT must be extended by carefully defining the new keywords and adding them to the XMT repository. Using reserved keywords for context, problem, and solution allows for automated text processing, a scalable solution. However, the three items of the MT will have to be divided into a "brief" part, containing the reserved keywords, and an elaborate part containing the explanation and details of each item. This makes it suitable for human reading as well as machines, and is sometimes applied to the solution item by providing a thumbnail solution "solution-brief" and an elaborate solution.

- **Definition**: *Similarity Criteria* is a set of logical conditions that decide if a pattern A is similar to a pattern B.
- **Definition**: *Similar Patterns* (a similarity criterion)

  Pattern A is similar to pattern B, denoted A $\equiv$ B, if[2]

  MT (A) = MT (B)

  (where MT denotes the minimal triangle)
- **Definition**: *Equivalent Patterns* (a similarity criterion)

  Pattern A is equivalent to pattern B, denoted A $\approx$ B, if

  Problem (A) = Problem (B) and
  Context (A) = Context (B)

We can see that an equivalence relationship is a superset of a similar relationship.

Two issues arise:

- Identical patterns are to be determined by pattern author (by including aliases to a pattern either to refer to another known name, or another pattern), but will also be complemented by our Structured Expert Support (SES) as shown in [2].
- Similar and equivalent patterns can be determined within the activity of SES in two different ways:
  - **Formally**, by comparing the three MT items and applying the definitions of similarity given above. Once patterns are modulated according to the XMT model, this process can be automated using simple tools.
  - **Informally**, by manually selecting patterns according to SES similarity and redundancy criteria.

[2] As in 1, this is a sufficient, but not a necessary condition

2. Structured Expert Support Implementation

Beside models, manual work is essential to both clearing out, and uploading modeled patterns. Here we create new assumption set, and define some activities to help build the new system. To address the relationships between patterns as explained in the extrinsic data part within the GPT [3], we have defined two types of relationships:

- **Structural relationships**: These are patterns that have some common structure (like a common MT; or part of it).
- **Assimilation relationships**: Those patterns are considered from the design process point of view. Two patterns that are completely different in structure can still have an assimilation relationship, like complementing or competing with each other.

2.1. Argument

These two types might look orthogonal to each other, but -generally speaking- we assert that structural relationships should be included as a subset of all legitimate assimilation relationships. During the design process, patterns can be replaced based on several criteria. Similarity of pattern structure is a criterion that warrants the possibility of replacement. This concept will be demonstrated with some examples later in the paper. We will discuss this further with the issue of redundancies and show the unwanted effect of structurally entwined patterns.

Gamma et al. [5] emphasize in their book "Design Patterns" that defining the contextualized relationship between patterns is a key notion in the understanding of patterns and their usages. Zimmer [8] implements this idea by dividing the relations between the patterns of the Gamma catalog itself in 3 types: "X is similar to y", "X uses Y", and "Variants of X uses Y". Based on Zimmer's work, we showed in [4] some additional relationships between patterns from an assimilation point

of view, not a structural one.
We restate them here:

- **Subordinate (X, Y)** if and only if X is embeddable in Y. Y is also called superordinate of X.
- **Equivalent (X, Y)** if and only if X and Y can be replaced by each other.
- **Competitor (X, Y)** if X and Y cannot be used at the same time for designing the same artifact.
- **Neighboring (X, Y)** if X and Y belong to the same pattern category (family) or to the same design step as the described pattern.

A major nuisance to our work on pattern relationships, and certainly that of other users is the *"noisy similarities"* between patterns. We can visualize the relationships between patterns as in figure 3. Part of the similarity is a healthy relationship of **similar** or **equivalent** patterns that can easily replace each other; represented by the intersection area of the two ellipses. Several dissimilar patterns have relationships like **competitor**. A considerable part of similarities, however, has redundancies in it, and we excluded them from the concept of pattern relationships. They are represented in the part of the "pattern similarities" ellipse outside the relationships ellipse. According to our investigation on many patterns, it is not easy to decide on the nature of the relationships in these cases.

To demonstrate, if we said that pattern A is a competitor to pattern B, and then we have another pattern C which is slightly (or -in general- vaguely) similar to pattern B, can we say that pattern C is also a competitor to pattern A. The answer is that we have to resolve the vague similarity between B and C to remove this ambiguity first. Logically speaking, this will not guarantee a solution to the question. But if we were able to assert -for example- that pattern B and pattern C have an inheritance relationship, we might assume that A and C are likely competitors as well. If we were able to assert that pattern B and pattern C are similar, then we can remove pattern C altogether. In many cases we found that some patterns contain a full detail of other patterns with a slight addition and often with a slight omission. Other similarities have more entwined structure that has no clear answer. We provide more discussion in the next section.

2.2. Inter-collection Redundancies ICR

Besides our automatic discovery approach through the XMT model, similar patterns have been discovered and recorded manually. These two approaches are going in parallel to complement each other as explained earlier. We have identified many identical patterns and similar patterns with varying degrees of similarity. In some cases we were able to identify patterns in different collections that are exactly the same, but were presented by different authors under different names, despite their identical analysis. This is an "easy work", and these are marked simply for removal. Other identical patterns are more difficult to locate.

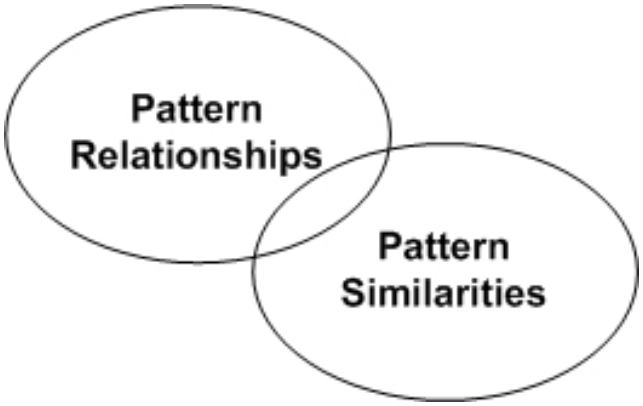


Figure 3. Useful and noisy similarities

The challenge comes with what we define as *entwined patterns*. They represent another case of partially similar patterns that can be confusing and has to be resolved. It is when two (or more) patterns have common features but each one still has a unique set of features. Despite the several types of entwining -depending on the type of redundant features- they can all be logically represented by the case of entwined brackets:

$$\{PartOfPatternA(CommonPart\}PartOfPatternB)$$

In several domains, like in programming languages, this is generally not allowed. In pattern domain, our observations show that this is one of the most confusing redundancies between patterns. We need to dissolve these patterns either by combining them into one pattern or separating them into two distinct patterns.

Note: Analogous to the defined relationships of inheritance (is-a) and aggregation (has-a) in object oriented programming, we also have pattern inheritance and aggregation relationships. In patterns domain, these are not a problem of similarity or a source of confusion; the former is addressed in a hierarchical model. The latter is a clear case of assimilation relationship. Connecting these patterns together is done through the extrinsic data part of the GPT.

3. Conclusion

In this paper we addressed the problem of redundancies between patterns and its negative effect on correct pattern reuse. It is hard and impractical to ask pattern users to observe and avoid redundancies among them. A more feasible approach is to detect and analyze these redundancies and try to reduce them. We selected interaction design patterns as a start and collected many of them. After detailed comparison and analysis, we identified several types of redundancies ranging from identical patterns (100 percent similar) into lower degrees of similarity and we grouped them into different types. We then formally defined these types in abstract formats to help reapply them on other types of patterns. We used auxiliary models to help apply the similarity criteria in an automated way as well as manually. In the future, we are extending our work on two fronts:

1. We are growing the eXtensible Minimal Triangle to add more standardized keywords to help automate the detection process of pattern similarities/redundancies.

2. We are applying the abstract concept of redundancies in other domains of software patterns.

References

[1] S. Fincher and J. Finlay. Chi 2003 report: Perspectives on hci patterns: Concepts and tools; introducing plml. *Interfaces, the international journal of human computer interaction*, 56:26, Autumn 2003.

[2] Ashraf Gaffar. The 7c's: An iterative process for generating pattern components. In *proceedings of HCI International, the 11th International Conference on Human-Computer Interaction*, 2005.

[3] Ashraf Gaffar, Ahmed Seffah, and John Van der Poll. Hci patterns semantics in xml: A pragmatic approach. In *proceedings of of HSSE '05, International workshop on Human and Social Factors of Software Engineering, in ICSE 2005, the 27th International Conference on Software Engineering,*. ACM publishing, New York, NY, US, 2005.

[4] Ashraf Gaffar, Daniel Sinnig, Ahmed Seffah, and Peter Forbrig. Modeling patterns for task models. In *proceedings of TAMODIA, 3rd International Workshop on Task Models and DIAgrams for user interface design*, 2004.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994. ISBN: 0-201-63361-2.

[6] Charles F. Goldfarb and Paul Prescod. *Charles F. Goldfarb's XML handbook*. Charles F. Goldfarb definitive XML series. Fifth edition, 2004. ISBN: 0-13-049765-7.

[7] G. Meszaros and J. Doble. A pattern language for pattern writing, Mai 5th 1997.

Available at: `http://hillside.net/patterns/writing/patternwritingpaper.htm`.

[8] Walter Zimmer. Relationships between design patterns. pages 345–364, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co. ISBN: 0-201-60734-4.

Evaluating the Use of Design Patterns during Program Comprehension – Experimental Setting

Yann-Gaël Guéhéneuc and Stefan Monnier
Department of Informatics
and Operations Research
University of Montreal, Quebec, Canada
{guehene,monnier}@iro.umontreal.ca

Giuliano Antoniol
Computer Engineering Department
Polytechnic Montreal, Quebec, Canada
antioniol@ieee.org

Abstract

In theory, there is no difference between theory and practice. But, in practice, there is.

Jan L. A. van de Snepscheut

1 Introduction

Design patterns [2] have been adopted quickly by the software engineering community and, since their introduction, several studies have been proposed to ease their choice [5], their use [1], and their recovery [10].

A design pattern is a literary form providing solution to recurring design problems. It decomposes in several sections: Intent, Motivation, Applicability, Structure, Participants, Collaborations, Consequence, Implementation, Known Uses, and Related Patterns. The solution advocated by a design pattern, mainly in the Structure, Participants, and Collaborations sections, is a design motif—a prototypical micro-architecture that describes the solution while abstracting context and implementation constraints.

Many studies (including studies by the authors) claim that identifying micro-architectures similar to design motifs in program architecture benefits software engineers by easing program comprehension. The rationale of the studies on design motif identification is:

- Design decisions are scattered in a program architecture and are often not documented.
- Design decisions help software engineers in understanding the design of a program.
- Micro-architectures similar to design motifs are clues on the design decisions.

Although the previous claim is logical and reasonable, some studies show that the use of design patterns does not ease program comprehension and, thus, question the benefits of design motif identification. In particular, the Wendorff's qualitative study [9] raises the issue of overuse and misuse of design patterns.

We believe that, indeed, design motif identification benefits software engineers by easing program comprehension but seek proofs of this claim. Thus, we suggest experimental settings with which to prove or to disprove the claim that design motif identification help software engineers' comprehension.

In Section 2, we elaborate on the research question related to the claim. In Section 3, we propose experiments to test our research question as well as experimental settings. Finally, in Section 4, we conclude on the research question and the experiments.

2 Research Question

2.1 General Question

Question. Our research question concerns the evaluation of the help that design motif identification really provides to software engineers during program comprehension.

Rationale of the Question. The cost of maintenance is evaluated to at least 50% of the overall cost of software development. Among those 50%, another 50% (for the least) is dedicated to program comprehension. Hence, any help in comprehending programs could reduce the cost of software development dramatically.

Many studies claim that design patterns, in general, ease program comprehension and that the documentation of used design motifs (possibly through semi-

automated identification), in particular, allows software engineers to grasp design decision quickly.

However, to the best of our knowledge, no systematic studies have been published yet to prove or to disprove the claim that design motif identification help in comprehending programs.

2.2 Hypotheses

Acquisition of the Information. Software engineers, like people in general, use their sight as the main mode for acquiring information. Indeed, software engineers spend long hours looking at computer-displayed models of programs in various forms and dimensions.

A well-known way of displaying program models is using the UML notation: 2D diagrammatic models of different aspects of a program (structure, behaviour, packaging...). Figure 1(a) shows a UML-like model of a sample program.

The use of sight suggests that many factor must be considered to understand how software engineers comprehend a program using a model such as shown on Figure 1(a). In particular, proximity, attention, object recognition and categorisation impact the quality and pertinence of the acquired information.

In the case of design motif identification, the use of visual models also raises the issue of the ease to identify visually design motifs with no a priori clues.

Use of the Information. Once design motifs have been identified (either visually or by other means), software engineers can use this information to comprehend a program better. They may use this information either to focus their attention towards the classes, methods belonging to an identified design motif (see Figure 1(b)), or to exclude from their attention those classes and methods to focus on other part of a program architecture (see Figure 1(c)).

2.3 Experimental Questions

How do software engineers look for design motifs? This question concern the way software engineers navigate through a program architecture when no design motif have been identified, documented, and displayed. Do the software engineers look for design motifs? Do they follow some specific path?

Do software engineers focus their attention on identified design motifs? This question focus on the use of identified and displayed design motifs over a classic model of a program architecture. Do software engineers focuses on the constituent of the design motifs or—on the opposite—focus on constituents of the program architecture away from the identified design motifs?

When do software engineers need to comprehend a program? What do they need to comprehend? Although not directly related to design patterns, these questions are clearly related to the two previous questions. It is most relevant to understand the software engineers' comprehension activities to contextualise the two previous question. Indeed, the context is important to set up relevant experiments. Unfortunately, to the best of our knowledge, no thorough studies of the software engineers' comprehension activities exist. An interesting study has been proposed by Murphy *et al.* [6] to analyse development activities. We hope this study will bear fruits that will help us in setting up our experiments.

3 Experiments

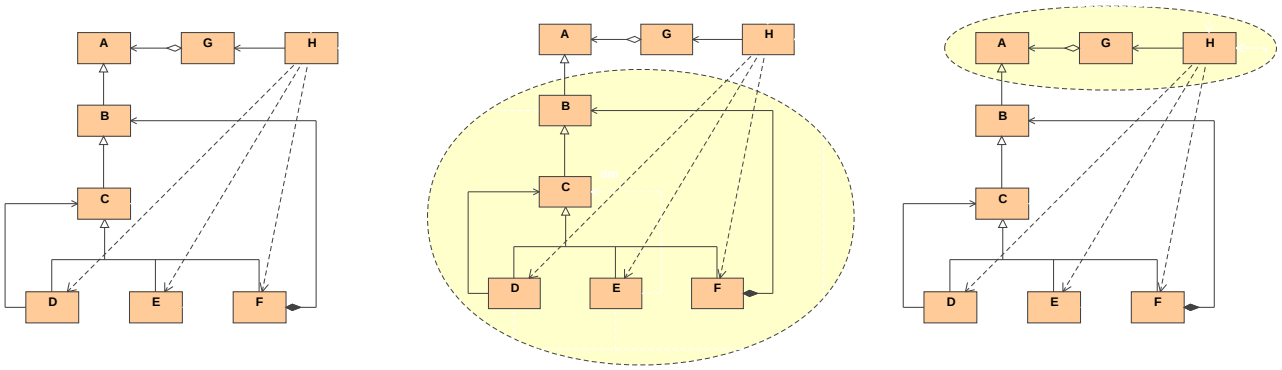
The use of new technology can help in setting up experiments to answer our experimental questions. We devise experiments to assess the identification of design motifs, on the one hand, and the use of identified design motifs, on the other hand, during the software engineers' program comprehension activity.

3.1 General Setting

Progress in non-intrusive monitoring of human behaviour makes it possible to follow the external behaviours of a software engineer involved in program comprehension activities without disturbing *too much* the performed activities.

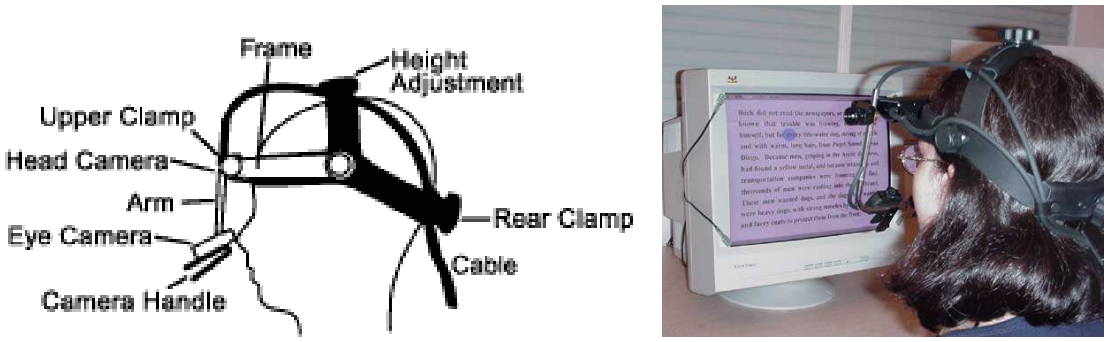
In particular, the use of video-based eye tracking systems allows following with enough precision a software engineer's eye movements while looking at a model of a program. A video-based eye tracking system records a subject's eye movements without much interferences with the subject's activity through the use of a special headband and an dedicated API.

In our experiments, we plan to adopt SR Research eye-tracking systems. SR Research is an international manufacturer of high quality eye-tracking systems with their EyeLink II systems. A EyeLink II system decomposes in a display computer, displaying the data to comprehend by the software engineer, a host computer storing the data related to a subject's eye movements, and a headband supporting cameras to track the eye movements.



(a) Sample UML-like diagram. (b) Focus on the Composite design motif. (c) Focus away from of the Composite design motif.

Figure 1. Sample UML-like diagram, with visual attention focused on or away from a micro-architecture similar to the Composite design motif.



(a) Headband and its different parts. (b) Use of the headband.

Figure 2. The SR Research EyeLink II eye-tracking system.

Figure 2(a) depicts the headband and its different parts, while Figure 2(b) shows a subject wearing such a headband. The headband mainly consists of a set of cameras recording the position of the head and the movements of the eyes with respect to the displayed image. Synchronisation with the image displayed on screen (being looked at by the subject) is perform through a dedicated API, which controls and synchronises the camera and generates the data. The generated data is stored on a dedicated host computer, hosting the eye-tracking system, and connected with the display computed by a high-speed ethernet connection. Figure 3 summarises the data acquisition process.

We plan to use such a system to study the eye movements of the software engineers on the constituents of class diagram-like models of programs and the dwell time on individual constituents such as class, relationships, identified design motifs. We shall adapt the PTIDEJ tool suite to synchronise the display of class diagrams-like models with EyeLink II systems. PTIDEJ [3] is a set of tools to evaluate and to enhance the quality of object-oriented programs, promoting patterns, at the language-, design-, and architectural-levels. With PTIDEJ, it is possible to create and to display models of AOL, C++, and Java programs using a unified meta-model, to infer inter-class relationships [4], and to identify structural design motifs.

3.2 Experimental Setting

We want to understand how software engineers use design motifs during program comprehension activities. Thus, the subjects of our experiments are software engineers while the objects are models of programs highlighting or not design motifs.

Consequently, in the perspective of generalising the results of our experiments, we must choose our subjects carefully. We must choose subjects and form two

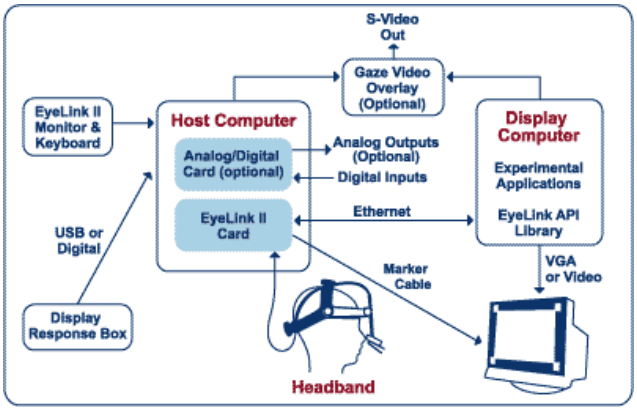


Figure 3. Use of a EyeLink II system.

groups: Subjects with no knowledge of design patterns and subjects with a deep knowledge of design patterns. We must perform our experiments with sufficient numbers of subjects in both group to avoid experimental biases. We plan to use students at the Department of Informatics and Operations Research as subjects. First or second year bachelor students would belong to the group with no knowledge of design patterns, while Master or Ph.D. students would belong to the group with good knowledge of design patterns.

The objects of our experiments must be programs in which developers use design patterns. However, we can limit ourselves to a small set of programs providing that we limit the reinforcement learning process [8]. We plan to use the JHotDraw and JUnit programs because those are well-know programs with limited numbers of classes and an average complexity, in which developers used design patterns.

The program comprehension activities we shall ask software engineers to perform must involve both part of the program architectures related to design motifs and not related to design motifs. A typical task involving a design motif would be to add a class to the design motif (*i.e.*, a functionality to the program). A typical task not involving a design motif would be to modify a class to change the program behaviour.

3.3 Identification of Design Motifs

Hypothesis. We make the hypothesis that expert software engineers look for micro-architectures similar to design motifs during program comprehension while novice software engineers do not.

Expected Results. The expected results of the experiments is that expert software engineers spend time, at the beginning of the program comprehension activity, in searching for design motifs, while novice software engineers just follow in rand orders the relationships among classes to perform their tasks.

3.4 Use of Identified Motifs

Hypothesis. Our hypotheses is that the knowledge of the design motifs used in the design of a program decrease the time for program comprehension. This knowledge allows software engineers to focus on constituents in the micro-architectures highlighted as design motifs or away from these constituents.

Expected Results. Depending on the task at hand and the knowledge (or lack thereof) of the existing design motifs, we expect to measure the difference of time spend by expert and novice software engineers on the constituents of micro-architectures similar to design motifs. We expect that expert software engineers will use their knowledge and the knowledge of the used design motifs.

Discussion. Our hypothesis is similar to the idea of "intentionally ignored information", such as experimented by Rock and Gutman [7].

4 Conclusion and Future Work

We presented experimental settings using eye-tracking systems to understand how software engineers (novice and expert) use knowledge of the design patterns used in the design of programs.

We believe these experimental settings will help in understanding the use of design patterns and answer questions related to their wide adoption and their far-reaching use in practice.

We now plan to refine these experimental settings and to perform the actual experiments as part of the development of the LaiGLE laboratory (Laboratory for Experimental Software Engineering).

References

[1] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, February 1998.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[3] Yann-Gaël Guéhéneuc. Ptidej: Promoting patterns with patterns. In *proceedings of the* 1^{st} *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005. **Submitted for publication.**

[4] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *proceedings of the* 19^{th} *conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, October 2004.

[5] Olivier Motelet. An intelligent tutoring system to help OO system designers using design patterns. Master's thesis, Vrije Universitët, 1999.

[6] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Čubraniś. The emergent structure of development tasks. In Andrew P. Black, editor, *proceedings of the* 19^{th} *European Conference on Object-Oriented Programming*, pages 33–48. Springer-Verlag, July 2005.

[7] Irvin Rock and Daniel Gutman. The effect of inattention on form perception. *journal of Experimental Psychology: Human Perception and Performance*, 7:275–285, 1981.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1*st* edition, March 1998.

[9] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and Jürgen Ebert, editors, *proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

[10] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *proceedings of the* 26^{th} *conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.

Enhancing Software Evolution with Pattern Oriented Software Product Life Cycle

Jayadev Gyani
University of Hyderabad
INDIA
jayadevgyani@yahoo.com

P.R.K. Murti
University of Hyderabad
INDIA
prkmcs@uohyd.ernet.in

Abstract

In the current age of software development design patterns have to play a greater role in software development. Design patterns can be defined as the solutions to the frequently recurring problems in design. Pattern based development will improve the software reusability if properly utilized. In this paper, we present pattern oriented software product life cycle (PSPLC) with a focus on Gang-of-Four design patterns. Many developers are resisting the use of design patterns because of unfamiliarity with this area. This approach is very useful to novice developers. Software evolution implies modifying requirements or adding new requirements. PSPLC thrusts the developers in using design patterns, which results in improved software evolution. A case study on real time scheduling is discussed for showing the applicability of PSPLC.

Key words: Design patterns, product life cycle, software design reuse, software evolution

1.Introduction

Using the idea of design patterns early in the product life cycle will improve the clarity of the software architecture. If design patterns are suggested before going to the design phase, the new developers can throw light on using design patterns. Even though using a design pattern for a specific set of requirements is a creative task, this approach aids in understanding the use of design patterns. In section 2, we propose Pattern Oriented Product Life Cycle. In section 3, we present few guidelines to suggest design patterns based on the specification.

2. Pattern oriented product life cycle

General product life cycle contains the analysis, design, coding, testing and installation phases. We suggest a new phase called ***pattern suggestion phase*** before design phase. **Figure 1** shows the new pattern-oriented software product life cycle. We call it as PSPLC.

It starts with the object oriented analysis phase encompassing the specification of classes, methods and class relationships. Any of the standard object oriented analysis methods can be used. The specification should include the features for representing association, subclassing, composition and other relationships.

In the next proposed phase, design patterns can be suggested based on the requirement specification. The specification can be analyzed either manually or using a simple parser. The criteria for suggesting patterns will be discussed in section 3. This phase can be called as pattern suggestion phase.

In the design phase, the suggested patterns can be analyzed in the context of the current application. A set of guidelines will motivate the developer to look through these patterns. When the suggested patterns are not suitable to the current application, new design patterns can be designed and added. It is an evolving phase. All the benefits of using design patterns may not be realized for the first time.

When the designs are implemented in one of the languages, the application can be tested. After successful testing and installation, new design patterns can be mined from the current working application. The criteria for suggesting these patterns can be added to the pattern suggestion phase.

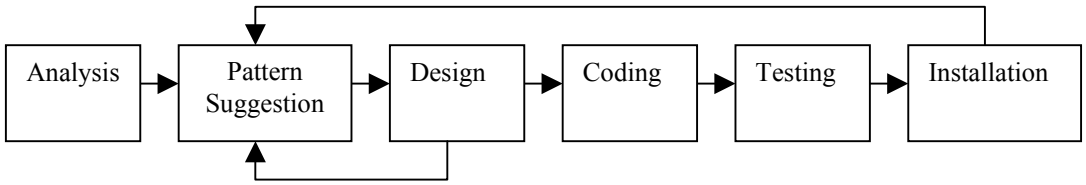


Figure 1. Pattern oriented product life cycle

3. Criteria for suggesting patterns

Design patterns are solutions to recurring problems in design[1]. Design patterns can be used for creating flexible and maintainable software. Extracting design patterns from design or code is suggested by G. Antoniol et al.[3]. Design patterns are used in developing application frameworks[4]. Pattern oriented software development life cycle model was suggested by M.S.Rajasree et al.[5]. Their approach was based on creating global structure based on communication model. This model focused on transforming requirements to design patterns.

Natural language heuristics can be used as guidelines for suggesting patterns. These guidelines are the minimum evaluation criteria for suggesting a pattern from the specification. If the suggested patterns are not suitable to the current application, the life cycle proceeds in a normal way. Little material is available on suggesting patterns using natural language heuristics. These guidelines are useful when the requirements are specified in natural language. Criteria for suggesting six design patterns will be discussed in this section. All these patterns are taken from Gang-of-Four design patterns.

3.1 Adapter

The structure of the adapter pattern is shown in **Figure 2**.

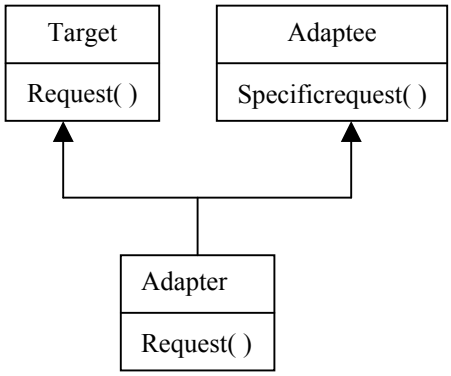


Figure 2. Adapter

Criterion 1: "Must comply with existing interface" or "Must reuse existing legacy component".

3.2 Bridge

The structure of the bridge pattern is shown in **Figure 3**.

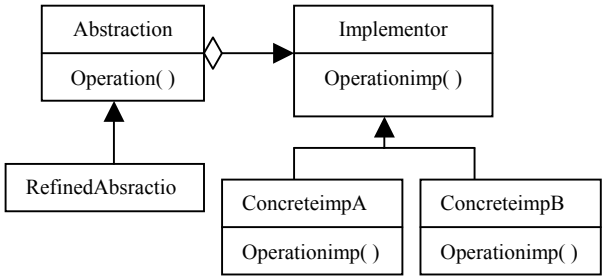


Figure 3. Bridge

Criterion 2: "Must support future protocols."

3.3 Composite

The structure of the composite pattern is shown in **Figure 4**.

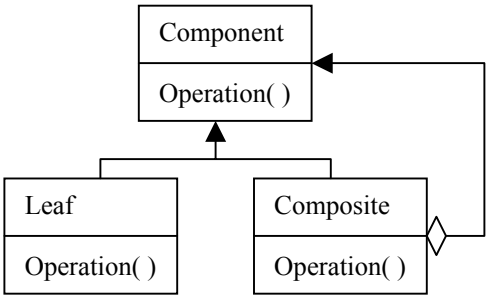


Figure 4. Composite

Criterion 3:"Must support aggregate structures."

3.4 Strategy

The structure of the strategy pattern is shown in **Figure 5**.

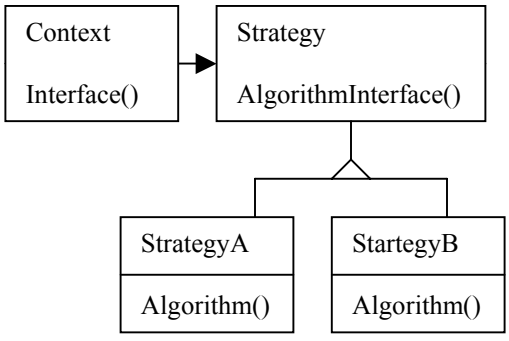


Figure 5. Strategy

Criterion 4:"Must implement different algorithms and a common interface to be provided"

3.5 Façade

The structure of the facade pattern is shown in **Figure 6**.

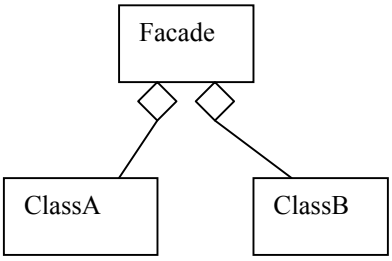


Figure 6. Facade

Criterion 5: "Must provide a unified interface for various subsystems."

3.6 Proxy

The structure of the proxy pattern is shown in Figure 7.

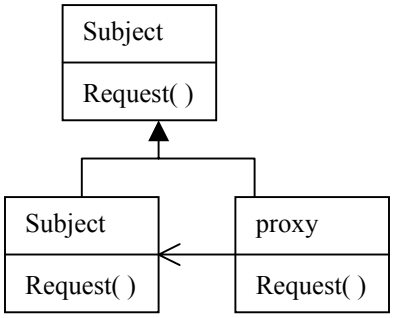


Figure 7. Proxy

Criterion 6:"Must support simple object creation when the actual object creation is time consuming."

4. Case Study

We explain our concept with Real time scheduling. Requirements Specification for real time scheduling in plain English will be as follows:

1.Real time scheduling algorithms for periodic task requires various parameters such as phase, period, execution time and relative deadline.

2. Application should implement different algorithms with common interface. The user interface takes the parameters specified in the above requirement.

3.Rate-monotonic(RM) and Deadline-monotonic (DM) algorithms have to be implemented.

4.Other scheduling algorithms need to be added later.

5.Output shows the schedule diagram of selected algorithm.

Above Requirements seem to be simple but they are sufficient to explain our concept. According natural language heuristics, ***Strategy*** pattern can be used. When multiple patterns satisfy a heuristic, then different solutions can be suggested. Identifying a pattern before the design phase will help the developer in framing the solution effectively. This will be done during ***pattern suggestion*** phase. PSPLC emphasizes the use of design patterns, which results in enhanced software evolution. In the given example if any other algorithm is to be implemented such as Earliest-Deadline-First algorithm, then this can be done adding another concrete strategy class. Strategy pattern for real time scheduling can be shown in **Figure 8**.

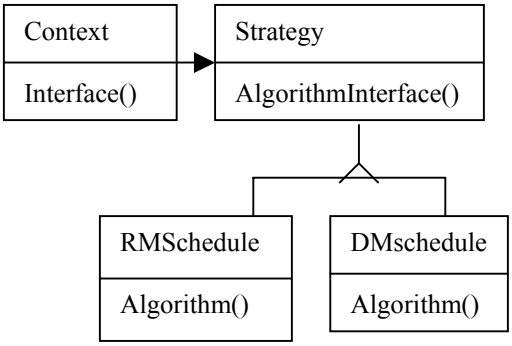


Figure 8. Strategy for Real Time Scheduling

After developing the application the user would like to implement one more algorithm EDF that can be simply added to the Strategy without affecting other modules. The resulting figure is shown in **Figure 9**.

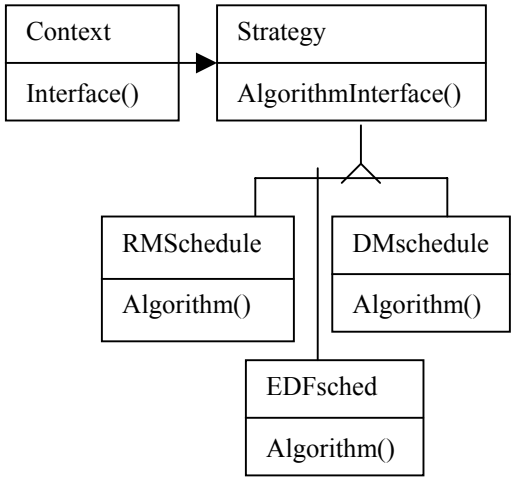


Figure 9. Strategy after adding EDF

After installing the application, new patterns can be mined such as user interaction pattern. We suggest one pattern for user interaction for real time scheduling. We name it as "RealSchedule". We have implemented this user interface in java. Whenever real time scheduling is required, this pattern can be used. A snapshot of this pattern is shown in **Figure 10**. Same user interface can be used for DM schedule also.

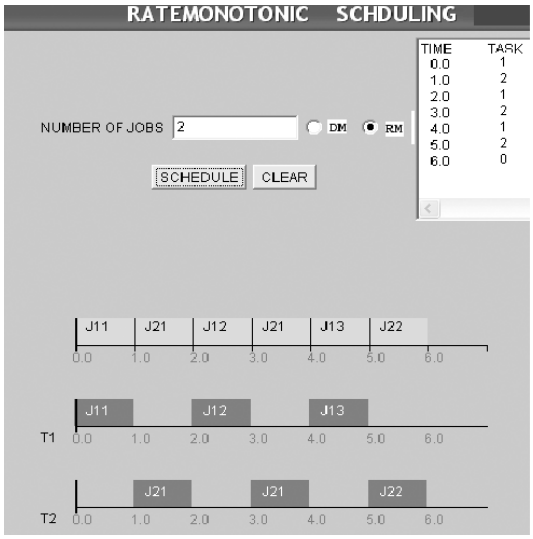


Figure 10. RM schedule

5. Conclusions and future work

Pattern-oriented product life cycle focuses on design reuse. While suggesting patterns, we focussed on the few patterns of GOF pattern catalogue because of simplicity in understanding the behavior of these patterns. However, same approach can be extended to include other patterns of GOF pattern catalogue and POSA patterns.

6. References

[1]. Erich Gamma and others, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1995

[2] Tiffany Winn, Paul Calder,"Is This a Pattern?", IEEE Software, January 2002, pp. 59-66

[3] G. Antoniol, R. Fiutem, L. Cristoforetti, "Using Metrics to Identify Design Patterns in Object-Oriented Software", 5th. International Symposium on Software Metrics, March 1998, pp. 23

[4] Mohamed Fayad and Douglas C. Schmidt, "Object-Oriented Application Frameworks", 1997, Communication of the ACM, Volume 40, Number 10, pp. 32-38

[5] M. S. Rajasree, P. Jithendra Kumar Reddy, D. Janaki Ram, "Pattern Oriented Software Development: Moving Seamlessly from Requirements to Architecture", International Workshop on Software Requirements to Architectures (STRAW 03) in association with International Conference on Software Engineering (ICSE 03) MAY 2003 Portland, Oregon, USA

A Taxonomy and a First Study of Design Pattern Defects

Naouel Moha, Duc-loc Huynh, and Yann-Gaël Guéhéneuc
PTIDEJ Team
GEODES - Group of Open and Distributed
Systems, Experimental Software Engineering
Department of Informatics and Operations Research
University of Montreal, Quebec, Canada
E-mail: {mohanaou,huynhduc,guehene}@iro.umontreal.ca

Abstract

Design patterns propose "good" solutions to recurring design problems in object-oriented architectures. Design patterns have been quickly adopted by the Software Engineering community and are now widely spread. We define design pattern defects as occurring errors in the design of a software that come from the absence or the bad use of design patterns. Design pattern defects are software defects at the architectural level that must be detected and corrected to improve software quality. Automatic detection and correction of these software architectural defects, which suffer of a lack of tools, are important to improve object-oriented architectures and, thus, to ease maintenance. We propose a first taxonomy of design pattern defects and presents techniques and tools to detect these defects in source code.

Keywords: Software Defects, Design Patterns, Design Pattern Defects, Antipatterns, Detection, Correction, Object-Oriented Architecture.

1. Introduction

Validation and verification as well as maintenance are key activities in the software lifecycle. During these activities, it is important to check the correctness of the design and implementation of a software product against some predefined criteria to detect and to correct software architectural defects early in the development process and, thus, to reduce costs.

We define design pattern defects as a sub-category of software architectural defects (SAD) [8]. Design pattern defects are distorted forms of design patterns, i.e., micro-architectures similar but not equal to those proposed by the solutions of design patterns, also called design motifs [4].

Long-term Research Objective. By detecting design pattern defects, it is possible to pinpoint which design patterns have not been well implemented by the developers and, thus, to understand why design patterns are not well applied. We are convinced that the identification of micro-architectures similar, but not equivalent, to design patterns not only highlight poor design solutions needing improvements, but also help in improving the application of design patterns. Through our research, we want to show which design patterns are usually well applied and which are less well applied, but also how they are applied and for which applications (domains, size, etc.). We believe that the detection and study of design pattern defects help to answer the questions: When, how, why and which design patterns are not well applied?

Short-term Research Objective. This paper proposes a first classification of the design pattern defects and presents tools and methodologies to detect them in source code. In section 2, we define what we mean by design pattern defects and propose a first taxonomy of these defects. In section 3, we present the current techniques and tools for detecting these defects. In section 4, we detail the case study we conducted to identify design pattern defects.

2. Terminology

A clear understanding of the different types of design pattern defects and a classification of those defects is

necessary before proposing any techniques related to their detection.

2.1. Taxonomy

This section clarifies what we mean by design pattern defects and proposes a first classification of this category of defects.

Defects. We use the term "defect" to define a flaw or an imperfection, not an erroneous behaviour. It is any unintentional, intentional, or undesirable irregularity in the design or the architecture that could affect performance and maintenance. A defect can lead to a fault that may cause a failure.

Design Pattern Defects. Design pattern defects are similar to design patterns which are today used and studied in the industry and academia. Design patterns propose "good" solutions to recurring design problems in object-oriented architectures, whereas design pattern defects are occurring errors in the design of the software that come from the absence or the bad use of design patterns. Thus, Guéhéneuc *et al.* define design defects, referring to design pattern defects, as distorted forms of design patterns, i.e., micro-architectures similar but not equal to those proposed by solutions of design patterns [4].

We propose to distinguish between four kinds of design pattern defects:

- *An approximative or deformed design pattern* is a design pattern that has not been well implemented according to the Gang of Four but that is not erroneous. Sometimes it can be an improvement of the implementation of the design pattern.
- *A distorted or degraded design pattern* is an incorrect occurrence or a distorted form of a design motif which is harmful for the quality of the code.
- *A missing design pattern* is one of the subcategories of design flaws defined by Marinescu [7]. According to the Gang of Four, missing patterns generate poor design.
- *A excess design pattern* is related to the excessive use of design patterns [12].

Thus, we define design pattern defects as design pattern motifs that are deformed, distorted, missing, or in excess, and that conflict with their motivations as in the Gang of Four. They are different from antipatterns because they were intended to improve design contrary to antipatterns that are bad solutions (we adopt the philosophical stance that too much of a good thing can be bad).

2.2. Classifications

It is important to have a consistent classification to avoid redundancies in definitions and to ease comprehension. We classify design pattern defects based on the two following classifications.

Gamma Classification. We propose the following classification, which describes the nature of design pattern defects, to maintain consistency between the classification of design patterns as defined by Gamma *et al.* [3] and design pattern defects:

- *Creational Design Patterns Defects* are defects related to creational patterns including Abstract Factory, Builder, Singleton, etc.
- *Structural Design Patterns Defects* are defects related to structural patterns including Composite, Decorator, Facade, etc.
- *Behavioral Design Patterns Defects* are defects related to behavioral patterns including Command, Iterator, Visitor, etc.

Classification of the type of defects. This classification aims to define not the nature but the kind of defects. Based on the different kinds of design pattern defects, we define four kinds of defects: missing, in excess, deformed, and distorted.

2.3. Origins of Design Pattern Defects

Software aging, as defined by Parnas [10], is one of the first explanation of the origins of design pattern defects. Indeed, the apparition of design pattern defects can be explained by the evolution of a program has been subject to a lot of changes: restructuring, additions, or removals of functionalities, of methods, of classes, etc. These changes may degrade the design of a program and, thus, the solutions of design patterns initially implemented.

The second explanation is the lack of experience of developers. Novice developers may not have an advanced knowledge of design patterns. They make an effort to structure well their programs by implementing design patterns but fail to implement some concepts related to design patterns or to restrain the use of design patterns. Others are simply not aware of these good practices implementations and use alternatives to solve well-known problems (for example, see the many possible solutions to the design pattern "Iterator").

3. Detection Techniques and Tools

The problem of automating the detection and correction of design pattern defects have not yet been largely studied.

Actually, the detection of design pattern defects is left to the intuition of the developers or architects based on their experience. But it is a tedious task, especially for large systems. The automatic detection of the design patterns requires to define specific and structured automatic techniques.

However, no technique or measure is as precise as the faculty of the developer to evaluate the code quality and most of the techniques generate some false positive detections. Thus, we suggest that the detection of design pattern defects must be semi-automatic.

The following techniques are currently used for the detection of design pattern defects. These techniques are split into three distinct categories: manual, semi-automatic, and automatic techniques.

- *Manual.* Manual techniques include the naïve approach that consists of reviewing comments and the name of classes, methods to get some keywords related to design patterns. The basic tool that can be used for applying the naïve approach is the textual search provided by most of the code editors to assert the presence of a design pattern defects based on the results of the textual search. This technique requires to have a strong knowledge on design patterns.

- *Semi-automatic.* Reengineering tools such as Describe [2], EclipseUML [9], Code Logic [6] prove to be helpful in the detection of design pattern defects. These tools do not detect automatically design patterns but reverse-engineer class diagrams, which facilitates the visual detection of patterns. When the code has been reverse-engineered, a methodology consists of identifying the abstract classes and interfaces and their inheritance relationships. Another methodology is based on the "Pattern Map" of the Gang of Four which gives the design pattern relationships. The existence of a pattern can suggest the existence of another pattern.

- *Automatic.* Automatic techniques include using Constraint Satisfaction Problem (CSP) as in the Ptidej tool suite [1], recently extended with numerical signatures [5] and techniques based on metrics. They define the problem of detecting a design pattern in terms of its variables, the constraints among the variables, and their domains. The set of constraints corresponds to the relationships among the entities defined by the design pattern. Marinescu detected design flaws by applying measurement-based rules on a system via his fully automatized ProDeOOS tool [7].

4. Case Study

We perform a first case study to prove the presence of design pattern defects. We ask bachelor students to look for design patterns, but not explicitly design defects, in an application. The object of the study is DrJava [11], a lightweight development environment for writing Java programs designed primarily for students. The subjects are two groups of 3 bachelor students with a good knowledge on design patterns.

Hypotheses. Our hypothesis is that DrJava, as a small program (version 2004: 413 classes for 28,522 LOC), is believed to contain design patterns because it is relatively well written.

We define several attributes to well identify the design pattern defects found:

- Defect Type can have several values: Missing, Deformed, Distorted, in Excess.

- Defect Fix: This attribute gives the refactoring solution to solve this defect.

- Defect Impact: This attribute gives the measure of the impact of the defect in terms of the effect on the system (performance, security, maintainability). The advantage is to present to the maintainers the impact of not correcting this defect. The impact can also related to the developers or the users in terms of usability.

- Defect Visibility : This attribute specifies if the defect can be revealed at a static or dynamic state i.e. if the defect is structural or behavioral.

Research Questions. The questions that we try to answer are: When, how why and which design patterns are not well applied?

Procedure. Three different versions of DrJava were analyzed to point out the changes between the different versions. The reverse engineering of DrJava was performed by the following tools: Describe, Code Logic, and Eclipse. The analysis was based on the semi-automatic methodology as defined previously and lasted around 2 weeks (i.e., 30 hours) for each group. The second step of this case study consisted of evaluating the correctness of the design patterns found.

Results. 38 design patterns were found by the two teams. As generally accepted, the use of design patterns increases the quality of an architecture. So, considering the relative important number of patterns, we could deduce that DrJava is well structured, and thus, confirm our hypothesis. Moreover, the comparison of three different versions of DrJava shows that the developers use the benefits provided by the design patterns (specially the State and Strategy design patterns) to extend the program. Table 1 gives the results of the design patterns found in DrJava. We notice the high use of the Singleton and the Template Method.

Among the 38 design patterns found, three were design pattern defects. An Iterator, a Memento, and a Command design patterns have not been well applied by the developers. All three of them were categorized as approximative or deformed design patterns which mean that they are not exactly design patterns as defined by Gamma *et. al* but are not erroneous and do not affect the quality of the code (see Table 2). These three design pattern defects have appeared in the first version and were not corrected in the two next versions. We can deduce that they were not harmful for the architectural quality and we categorize them as deformed defects.

Threats to validity. This case study was a first exploratory experiment, and thus, has several limitations. It is difficult to generalize our results because we did not get a significant number of design pattern defects. We planed to re-conduct the same case study but asking the students to explicitly list all design patterns found, including those that are not well implemented. This case study does not enable to give a response to the questions: When and why design pattern defects are not well applied? However, we detected which design patterns were not well applied and how. To be able to response to the questions when and why, a case study on several different applications is required.

5. Conclusion

Design pattern defects are defects that come from the absence or the bad use of design patterns. We define design pattern defects as design patterns that are deformed, distorted, missing, or in excess, and conflict with the motivations of the Gang of Four.

We presented techniques and tools to detect design pattern defects and a case study that illustrates the use of these techniques and tools.

The automatic detection of design pattern defects is an help provided to the developer, the maintainer, and the architect to facilitate their work. Our objective is to provide indications on the presence, the localization, and the nature of design pattern defects in programs. Moreover, thanks to the localization of defects in the implementation of programs, it is possible to understand and to explain the reasons for which the design patterns are badly implemented.

We hope that this first work will stir the interest of the participants of the workshop and we would appreciate any comments about this ongoing research.

References

[1] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In Debra Richardson, Martin Feather, and Michael Goedicke, editors, *proceedings of the* 16^{th} *conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, November 2001.

Available at: `www.yann-gael.gueheneuc.net/Work/Publications/`.

[2] Embarcadero. Describe, August 2005. Embarcaderos Describe is a UML design solution that provides your software development team with immediate visibility into your source code. The product adds a set of powerful visual tools for manipulating code, all directly within your existing development environment.

Available at: `http://www.embarcadero.com/products/describe/`.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994. ISBN: 0-201-63361-2.

[4] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Quioyun Li, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *proceedings of the* 39^{th} *conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.

Available at: `www.yann-gael.gueheneuc.net/Work/Publications/`.

[5] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Jean-Yves Guyomarc'h. Identification of approximate occurrences of design patterns: An experimental study. In John Knight, editor, *Transactions on Software Engineering*. IEEE Computer

Creational Patterns					Structural Patterns							Behavioral Patterns										
Abstract Factory	Builder	Factory Method	Prototype	Singleton	Adapter	Bridge	Composite	Decorator	Facade	Flyweight	Proxy	Chain of Responsibility	Command	Interpreter	Iterator	Mediator	Memento	Observer	State	Strategy	Template Method	Visitor
1	0	1	0	12	2	1	0	0	0	0	1	0	2	0	1	1	1	0	3	3	9	0

Table 1. Results of the Detection of Design Patterns in DrJava application

Design Pattern Defects	*Defect Type*	*Defect Fix*	*Defect Impact*	*Defect Visibility*	Comments
Semi-automatic Analysis of DrJava					
Iterator	Deformed	No Fix	No Impact	Static	Iterate the real list and not a clone of the list.
Memento	Deformed	No Fix	No Impact	Static	Fusion of the caretaker and the originator in the same class.
Command	Deformed	No Fix	No Impact	Static	Fusion of the client and the invoker in the same class.

Table 2. Results of the Detection of Design Pattern Defects in DrJava application

Society Press, 2006. **To be submitted for publication in June 2005.**

[6] Inc. Logic Explorers. Code logic, August 2005. CodeLogic is a revolutionary system for discovering and graphically representing the deep, internal logic of any Java code. Developers can simply point CodeLogic at any existing Java or C# project and immediately get an intuitive view of exactly how the code works.

Available at: `http://www.logicexplorers.com/products/codelogic/`.

[7] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. Ph.D. thesis, Politehnica University of Timisoara, October 2002.

Available at: `www.cs.utt.ro/~radum/papers.html`.

[8] Naouel Moha and Yann-Gaël Guéhéneuc. On the automatic detection and correction of software architectural defects in object-oriented designs. In *Proceedings of the* 4^{th} *ECOOP Workshop on Object-Oriented Reengineering*, July 2005.

[9] Omondo. Describe, August 2005. EclipseUML Free Edition is a visual modeling tool, natively integrated with Eclipse 3.1 and JDK 5. EclipseUML Studio Edition offers full support for UML diagrams, team work, data J2ee modeling and dynamic collaboration to any other plugins.

Available at: `http://www.omondo.com/`.

[10] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN: 0-8186-5855-X.

[11] the JavaPLT group at Rice University. Drjava, August 2005. DrJava is a lightweight development environment for writing Java programs. It is designed primarily for students, providing an intuitive interface and the ability to interactively evaluate Java code.

Available at: `http://drjava.sourceforge.net/`.

[12] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and Jürgen Ebert, editors, *proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

Available at: `www.computer.org/proceedings/csmr/1028/10280077abs.htm`.

The Role of Design Pattern Decomposition in Reverse Engineering Tools

Claudia Raibulet & Francesca Arcelli
Università degli Studi di Milnao-Bicocca,
DISCo – Dipartimento di Informatica, Sistemistica e Comunicazione
{raibulet, arcelli}@disco.unimib.it

Abstract: *The decomposition of design patterns into simpler elements may reduce significantly the creation of variants in forward engineering, while it increases the possibility of identifying applied patterns in reverse engineering. Key questions raise here: what should be design patterns decomposed in? How to recognize the application of design patterns by identifying their components?*

Currently, there are few forward and reverse engineering tools that exploit the decomposition of design patterns (e.g., FUJABA, SPQR). FUJABA is a forward and reverse engineering tool introducing sub-patterns *to reduce the dimension of the design pattern catalog and the complexity of the elements searched in the source code, as well as to improve the detection algorithm. SPQR is an automatic tool for design pattern recognition, which introduces an* elemental design patterns (EDPs) *catalog and a rule set based on sigma-calculus through which EDPs are defined and composed into design patterns.*

In this position paper we aim at focusing on the advantages and disadvantages of decomposing design patterns in sib-components and at introducing our research interests and projects related to this issue.

Introduction

The idea of decomposing design patterns into recurring elements has emerged both in the context of forward and reverse engineering. Such elements are called fragments [6], motifs [5], minipatterns [4], sub-patterns [10, 11], or elemental design patterns [13, 15]. As the variety of the names suggests there is little agreement what design patterns should be decomposed in. From the abstraction point of view, the sub-components of design patterns should be situated at the half-way between the source code and the high-level definition of design patterns. Sub-components of design patterns should reduce the generation of variants in forward engineering, and increase the rate of identifying applied patterns in reverse engineering.

Although decomposing design patterns into sub-components may improve significantly their detection process and results, there are few tools (as far as we know) that exploit this approach: FUJABA (From UML to Java And Back Again) [10] and SPQR (System for Pattern Query and Recognition) [14]. The reason of decomposing design patterns into sub-components in the context of the two tools is different, however both obtain significant results. FUJABA is a forward and reverse engineering tool exploiting *sub-patterns* [11] to reduce the dimension of the design pattern catalog and the complexity of the elements searched in the source code, as well as to improve the detection algorithm. It builds a

hierarchy of sub-patterns by assigning them a level number which is exploited by the detection algorithm to establish the order of applying transformation rules. SPQR is an *automatic* tool for design pattern detection. *Elemental Design Patterns* (EDPs) [13, 16], the sub-components of design patterns, play a central role in the context of SPQR. Extraction of information from source code (currently only for C++) is performed according to the elements EDPs are built of. The design pattern detection is reduced to the EDPs detection, while design patterns are expressed exclusively through EDPs. EDPs and their composition rules are expressed formally in terms of rho-calculus [17], which represents a subset of sigma-calculus [1] extended with new reliance operators. Rho-calculus inherits from sigma-calculus the type definition, object typing, and type subsumption concepts. Reliance operators are defined as direct, quantifiable expressions which indicate whether elements rely or depend on other elements and to what extend they do so. These operators indicate reliance on method invocation, field access, or generalization. Note that EDPs form a plain abstraction level, this meaning they can be detected independently of each other. A detailed comparison of these two tools together with the advantages they provide is described in [2].

Advantages and Disadvantages of Decomposing Design Patterns

Considering the FUJABA and SPQR approaches, we summarize the advantages sub-components provide to define and detect design patterns:

- sub-components are less complex than design patterns, hence also the rules their detection is based on are simpler;
- design patterns can be described *formally* (according to SPQR) as combinations of their sub-components;
- decomposing patterns into sub-components do not affect the flexibility of the first once; the variants problem impacts mainly on sub-components;
- sub-components represent an intermediate abstraction level between design patterns and source code; they can be considered also independently from design pattern in the context of reverse engineering due to their ability of incorporating design intents.

Disadvantages are related to the initial stage of this research issue, hence the identification and specification of the sub-components of design patterns and how these sub-components are combined to form design patterns. In our opinion, there are design patterns that can be formed by the same set of sub-components, which are combined in different ways leading to different semantics.

Current and Future Work

Considering the SPQR approach particularly interesting, we have extended it to software systems written in Java [3]. We have used the Recoder [12] framework to parse the source code and to generate its equivalent AST. Further, we have implemented a Visitor [7] to extract the information given as input to the OTTER theorem prover used by SPQR and to encode this information in a POML (Pattern Object Markup Language) file [17]. The output of our prototype called J2POML [3] is given as input to the theorem prover of SPQR, which provides a report with the detected design patterns.

Further, we are evaluating the idea of using the catalog of EDPs independently of the SPQR approach. This choice considers also that SPQR is based on a mathematical paradigm not commonly used by software engineers, and difficult to be understood and extended for other programming languages or new design patterns. Moreover, SPQR is not available for testing.

We have implemented a prototype called EDPDetector4Java [9] which is able to identify EDPs within the Java source code. Each EDP is detected though two main types of information: (1) the relationship between the method (*referrer*) which calls another method (*referred*), and (2) the relationship between the object which contains the referrer (*sender*) and the object which contains the referred (*receiver*). The analysis of these two relationships is described in detail by the authors of SPQR in [15]. The goal of SPQR has been to generalize the analysis for object-oriented languages. Our aim is to particularize the analysis for the Java language. Each method invocation which characterizes the behavior of the objects defining an EDP is determined by both the relation between the objects and the relation between the methods involved in the interaction. We have identified eight methods' and objects' properties we can exploit in the Java language to define and detect all EDPs. In our approach, EDPs are described through functions expressed in terms of canonical sums of product forms of the eight properties. Our approach is definitely simpler than that of the SPQR, but it pays in precision. The results show that six of the sixteen EDPs are uniquely identified through a single combination of the eight properties, five EDPs through two combinations, three EDPs through four combinations, one EDP through six combinations, and one through ten combinations.

Currently, we have concentrated our attention only on static analysis of the source code. The limitations of such an analysis are generated by the polymorphism of the object-oriented languages. Our approach uses an AST representation of the source code in association with information related to type expressions, reference resolutions, and cross-references. Considering only the static analysis, we cannot detect EDPs precisely, hence they have associated a degree of uncertainty as in the FUJABA approach. We aim at overcoming this aspect by considering also the dynamic analysis of source code to determine the run-time type of the *receiver* and the *referred*.

Further work will follow two directions: (1) introducing dynamic analysis of the source code in order to uniquely identify each EDP through one combination of the eight Java properties, and (2) the identification of design patterns through combinations of EDPs. For the first objective we are studying the CAFFEINE [8] approach. Further, we have implemented the EDP catalog into FUJABA [18] to perform a deep comparison between the sub-patterns of FUJABA and the EDPs, comparison which may lead to a unified design patterns' sub-components catalog

References

[1] M. Abadi, and L. Cardelli, *A Theory of Objects*, Springer-Verlag, New York, Inc., 1996.

[2] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato. A Comparison of Reverse Engineering Tools based on Design Pattern Decomposition. In *Proceedings of the Australian Software Engineering Conference*, Brisbane, Australia, March, 28th-31st, 2005, pp. 262-269

[3] D. Bellinzona, "J2POML: Extraction of Information for Design Pattern Recognition from Java Source Code", University of Milano-Bicocca, Milan, Italy, November, 2004
[4] M. Ò Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. Ph.D Dissertation, University of Dublin, Trinity College, 2001
[5] A. H. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD Dissertation. Department of Computer Science, Tel Aviv University, 2000
[6] G. Florijn, M. Meijers, and P. van Winsen. Tool Support for Object Oriented Patterns. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, Springer Verlag, Berlin, Germany, 1997
[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: elements of reusable object-oriented software*, Addison Wesley, Reading MA, USA, 1994
[8] G. Y. Gueheneuc, R. Douence, and N. Jussien. No Java without Caffeine: A Tool for Dynamic Analysis of Java Programs. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, Semptember, 2002, pp. 117-126
[9] S. Masiero. Design Pattern Detection in Reverse Engineering – The Role of Sub-Patterns. Master Thesis, University of Milano-Bicocca, Milan, Italy, October, 2004
[10] U. Nickel, J. Niere, and A. Zündorf,. The FUJABA Environment. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, 2000, pp. 742-745.
[11] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-Based Design Recovery. In *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, 2002, pp. 338-348.
[12] Recoder - http://recoder.sourceforge.net/
[13] J. McC Smith. An Elemental Design Pattern Catalog. In *Technical Report TR02-040*, University of North Carolina at Chapel Hill, USA, December 10th, 2002.
[14] J. McC. Smith, and D. Stotts. SPQR: Flexible Automated Design Pattern Extraction From Source Code. In *Proceedings of the 2003 IEEE International Conference on Automated Software Engineering*, Montreal QC, Canada, October, 2003, pp. 215-224
[15] J. McC Smith, and D. Stotts. Elemental Design Patterns: A Link Between Architecture and Object Semantics. In *Technical Report TR02-011*, University of North Carolina at Chapel Hill, USA, March 25th, 2002.
[16] J. McC. Smith, and D. Stotts. Elemental Design Patterns: A Formal Semantics for Composition of OO Software Architecture. In *Proceedings of the 27th Annual IEEE/NASA Software Engineering Laboratory Workshop*, Greenbelt, MD, 2002, pp. 183-190.
[17] J. McC. Smith, and D. Stotts. Elemental Design Patterns and the Rho-Calculus: Foundations for Automated Design Pattern Detection in SPQR. In *Technical Report 03-032*, Computer Science Department, University of North Carolina at Chapel Hill, September 2003.
[18] D. De Bortoli, and L. Conti. *Recognition of Elemental Design Patterns in FUJABA.* BsC Thesis, University of Milano-Bicocca, Milan, Italy, April, 2005.

Workshop 4.2:
Workshop on System Integration and Interoperability

System of Systems Interoperability Issues

Dennis Smith, Edwin Morris, David Carney

Carnegie Mellon University

Software Engineering Institute

Abstract

A critical emerging software need concerns interoperability between systems and systems of systems. To meet this increasing demand, organizations are attempting to migrate existing individual systems that employ disparate, poorly related, and sometimes conflicting systems to more cohesive systems that produce timely, enterprise-wise data that are then made available to the appropriate users.

This paper first identifies a set of principles that need to be addressed to obtain interoperability and outlines a set of issues that need to be addressed in order to facilitate system of system interoperability.

1 Background

A critical emerging software need concerns interoperability between systems and systems of systems. This need is emerging within just about every software domain, including ebusiness, egovernment, integration of large military systems, mergers and acquisitions, and communications between embedded devices of systems that are traditionally considered to be hardware, such as automobiles and aircraft.

To meet this increasing demand, organizations are attempting to migrate existing individual systems that employ disparate, poorly related, and sometimes conflicting systems to more cohesive systems that produce timely, enterprise-wise data that are then made available to the appropriate users. Meeting this goal has often proven to be considerably difficult. This paper will focus on adoption centric issues that need to be addressed to obtain interoperability.

2 Background

A recent SEI study focused on identifying current problems in interoperability within DoD systems [Levine 02]. All of the problems were collected from interviews and workshops with personnel from the DoD.

The study indicated that new systems designed and constructed to interoperate with existing and other new systems, and adhering to common standards, still fail to interoperate as expected. Several of the general problems related to a need to have a better understanding of issues related to more effective adoption. These include:

- incomplete requirements
- unexpected interactions, and
- unshared assumptions

Specific interoperability problems included:

- Planned interoperability between new systems is often scaled back in order to maintain compatibility with older systems that cannot be upgraded without major rework.
- Strict specification of standards proves insufficient for achieving desired levels of interoperability, because organizations constructing "compliant" systems interpret specifications in different ways, thus creating different variants of the links.
- Policies promote a single point of view at the expense of other points of view. For example, policies that enhance the levels of interoperability that can be achieved in one domain are generalized to additional domains, where they unduly constrain organizations trying to produce interoperable systems.
- Funding and control structures in general do not provide the incentives necessary to achieve interoperability.
- Tests constructed to verify interoperability frequently fail to identify interoperability shortfalls. In other cases, systems are approved for release in spite of failing interoperability tests.
- Even when interoperability is achieved by systems of systems, it is difficult to maintain

as new versions of constituent systems are released. New system versions frequently break interoperability.

3 Guiding Principles

As Fred Brooks pointed out more than 15 years ago, the factors that make building software inherently difficult are complexity, conformity, changeability, and invisibility [Brooks 87]. Achieving and maintaining interoperability between systems is also inherently difficult, due to:

- complexity of the individual systems and of the potential interactions between systems
- lack of conformity between human institutions involved in the software process and resulting lack of consistency in the systems they produce
- changeability of the expectations placed on systems (particularly software) and the resulting volatility in the interactions
- invisibility of all of the details within and between interoperating systems

In spite of a considerable effort, technical innovations aimed at improving software engineering have not successfully attacked the problems represented by these essential characteristics. In fact, today's interoperating systems are likely more complex (due to the massive increase in the number of potential system-of-systems states) than those examined by Brooks. They exhibit less conformity (due to the increased diversity of the institutions involved in construction of the constituent parts), are more volatile (due to the need to accommodate widely diverse users) and have even poorer visibility (due to size, number of participating organizations, etc.)

We therefore posit a set of six principles that will inform our efforts in the selection of problems to address and, more critically, in the analysis of potential solutions. The principles are

1. No clear distinction exists between *systems* and *systems of systems.*
2. Most interoperability problems are independent of domain.
3. Solutions cannot depend on complete information.
4. No one-time solution is possible.
5. New technologies constantly move systems toward legacy status.
6. Networks of systems demonstrate emergent properties.

3.1.1 No Clear Distinction Between Systems and Systems of Systems

The distinction between a system and a *system of systems* is often unclear and seldom useful. By this we mean that many, perhaps a majority, of "systems" are actually systems of systems in their own right. The critical factor is less where a boundary might lie and more where control lies: most systems are now created with some components over which the integrator has less than complete control. Further, most systems must cooperate with other systems over which the integrator often has no control.

It is often stated that "What someone considers to be a system of systems, somebody else considers a system." Thus, for any given entity, one's perspective could see it as a component (of a larger system), as a system in itself, or as a system of systems. The Raptor's avionics system is certainly "a system." And, more importantly, *there usually is no top level*, because inevitably there will be some demand to include any system of systems in a more encompassing system of systems.

3.1.2 Interoperability Problems Independent of Domain

Most complex systems in almost every domain are now expected to interact with other complex systems. Regardless of domain, interoperability problems persist, and the costs of failures are huge. As an example, within the U.S. auto supply chain, one estimate put the cost of imperfect interoperability at one billion U.S. dollars per year, with the largest component of that cost due to mitigating problems by repairing or reentering data manually [Brunnermeier 99].

Our expectations are for even greater degrees of interoperability in the future, a goal that may prove difficult to achieve. The current generation of interoperable systems at least tend to be knowledgeable participants in the interaction—that is, the systems are being designed (or modified) specifically to interact with a particular system (or limited set of systems) in a controlled manner, and to achieve predetermined goals. What is new about the future generations of interoperating systems is an emphasis on dynamically reconfigurable systems. These systems—or more accurately the services they provide—are expected to

interoperate in potentially unplanned ways to meet unforeseen goals or threats.

We do not suggest that the solutions eventually found for the interoperability problems should be identical across domains. But we believe that the various communities should be aware of each other and look for commonality of high-level purpose and solution strategy—if not of solution detail—within other communities.

3.1.3 Solutions Cannot Rely on Complete Information

Classic software engineering practice assumes a priori understanding of the system being built, including complete and precise comprehension of

- assumptions or preconditions expected of the system that are required for successful use, including standards, system and environmental conditions, and data and interactions expected of other hardware, software, and users
- functionality, services, data, and interactions to be obtained from and provided to outside agents
- non-functional properties or quality of service required by the system and expected of the system from interacting components

For interoperable systems, the same information is required by all participants: the individual components (i.e., the individual systems), the links between them, and the composite system of systems. It would therefore seem that for an organization building a component (system), complete knowledge of all expectations is necessary to complete it. Unfortunately, we seldom (if ever) have such complete and precise specification even *when* a single system is *only* expected to operate in isolation.

The reality is that multiple organizations responsible for integrating multiple systems into interoperating systems of systems have multiple—and rarely parallel—sets of expectations about the constituent parts, as well as different expectations about the entire system of systems. The decisions that they make about the overall system of systems, e.g., assumptions, preconditions, functionality, and quality of service, are just as likely to be as incomplete and imprecise as those of organizations responsible for a single system.

Given that having complete and precise information about a system of systems (and its constituent parts) is not possible, two approaches to managing the potential chaos are evident:

- Reduce imprecision by enforcing common requirements, standards, and managerial control.
- Accept imprecision as a given and apply engineering techniques that are intended to increase precision over time, such as prototyping and spiral models of development.

The first approach alone may well increase interoperability to a significant degree, but it is also highly static and does not address the inherent imprecision in the software engineering process or the legitimate variation in individual systems. The second approach is limited in a different way, since without agreeing on some level of commonality, we are left with an "every system for itself" world that will not approach the levels of interoperability we require.

3.1.4 No One-time Solution Is Possible

We live in a dynamic and competitive world in which the needed capabilities of systems must constantly change to provide additional benefits, to counter capabilities of adversaries, to exploit new technologies, or in reaction to increased understanding or evolving desires or preferences of users. Simply put, systems must evolve to remain useful.

This evolution affects both individual systems and systems of systems. Individual systems must be modified to address unique and changing demands of their specific context and users. The expectations that systems of systems place on constituent systems will likewise change with new demands. However, the changing demands placed on a system by its immediate owners and those placed by aggregate systems of systems in which it participates are often not the same, and in some cases are incompatible.

The result is that maintaining interoperability is an ongoing problem. This was verified by our interviews with experts who had worked with interoperability. In some cases, desired system upgrades did not happen because of the impending effect on related systems. In other cases, expensive (often emergency) fixes and upgrades were forced on systems by changes to other systems.

In order to maintain interoperability, new approaches are needed to

- vet proposed requirements changes at the system and system-of-systems level
- analyze the effect of proposed requirements and structural changes to systems and systems of systems
- structure systems and systems of systems to avoid (or at least delay) the effect of changes
- verify interoperability expectations to avoid surprises when systems are deployed

New approaches to structuring systems that anticipate changes, that vet requirements and structural changes and analyze their impact, and that verify that systems of systems perform as anticipated will go a long way toward maintaining the interoperability of related systems.

3.1.5 Networks of Interoperability Demonstrate Emergent Properties

Emergent properties are those properties of a whole that are different from, and not predictable from, the cumulative properties of the entities that make up the whole. In very large networks, it is not possible to predict the behavior of the whole network from the properties of individual nodes. Such networks are composed of large numbers of widely varied components (hosts, routers, links, users, etc.) that interact in complex ways with each other, and whose behavior "emerges" from the complex set of interactions that occur.

Of necessity, each participant in such real-world systems (both the actor in the network and the engineer who constructed it) acts primarily in his or her own best interest. As a result, perceptions of system-wide requirements are interpreted and implemented differently by various participants, and local needs often conflict with overall system goals. Although collective behavior is governed by control structures (e.g., in the case of the networks, network protocols), central control can never be fully effective in managing complex, large-scale, distributed, or networked systems.

The net effect is that the global properties, capabilities, and services of the system as a whole emerge from the cumulative effects of the actions and interactions of the individual participants propagated throughout the system. The resulting collective behavior of the complex network shows emergent properties that arise out of the interactions between the participants.

The effect of emergent properties can be profound. In the best cases, the properties can provide unanticipated benefits to users. In the worst cases, emergent properties can detract from overall capability. In all cases, emergent properties make predictions about behavior such as reliability, performance, and security suspect.[1] This is potentially the greatest risk to wide-scale networked systems of systems.

4 Conclusion

We have outlined a set of principles that large scale systems of systems exhibit. These principles need to be more fully developed, and once fleshed out, they can become a starting point for research projects on interoperability and systems of systems. During the coming years, we expect that work will be done to

- codify current management and engineering practices for construction and evolution of systems of systems
- identify and characterize techniques for forming and evolving systems of systems
- develop evaluation approaches for selecting constituent elements of systems of systems
- analyze current and emerging technologies and products with potential applicability to the integration and interoperability of systems of systems
- develop cost models

References

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

[1] There is a continuing debate as to whether a property is truly emergent or whether, as we learn more about the components and their interactions, we will begin to find techniques for prediction of properties that appear to be emergent. We are neutral with regards to this debate, but believe that we must begin to understand the nature of emergent properties in networked systems of systems and find ways to manage problematic behaviors that occur—whether we can predict them or not.

Brunnermeier, Smita B. & Martin, Sheila A. *Interoperability Cost Analysis of the U.S. Automotive Supply Chain*. National Institute of Standards & Technology, March 1999. <http://www.nist.gov/director/prog-ofc/report99-1.pdf>.

Levine, L.; Meyers, C.; Morris, E.; Place, P.; & Plakosh, D. *Proceedings of the System of Systems Interoperability Workshop (February 2003)* (CMU/SEI-2003-TN-016). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tn016.html>.

Analyzing the Reuse Potential of Migrating Legacy Components to a Service-Oriented Architecture

Grace Lewis, Edwin Morris, Liam O'Brien, Dennis Smith
Software Engineering Institute
4500 Fifth Avenue,
Pittsburgh, PA 15213
{glewis, ejm, lob, dbs}@sei.cmu.edu

ABSTRACT

An effective way of leveraging the value of legacy systems is to expose their functionality, or subsets of it, as services. In the business world, this has become a very popular approach because it allows systems to remain largely unchanged, while exposing functionality to a larger number of clients through well-defined service interfaces. The U.S. Department of Defense (DoD) is also adopting this approach by defining service-oriented architectures (SOAs) that include a set of infrastructure common services on which organizations can build additional domain services or applications. When legacy systems or components are to be used as the foundation for these services, there needs to be an analysis of how to convert the functionality in these systems into services. This analysis should consider the specific interactions that will be required by the SOA and any changes that need to be made to the legacy components. The SEI has recently helped an organization evaluate the potential for converting components of an existing system into services that would run in a new and tightly constrained DoD SOA environment. This paper describes the process that was used and outlines several issues that need to be addressed in making similar migrations.

1. Introduction

With the advent of universal Internet availability, many organizations have leveraged the value of their legacy systems by exposing all or parts of it as *services*. A service is a coarse-grained, discoverable, and self-contained software entity that interacts with applications and other services through a loosely coupled, often asynchronous, message-based communication model [2]. A collection of services with well-defined interfaces and shared communications model is called a service-oriented architecture (SOA). A system or application is designed and implemented as a set of interactions among these services.

The characteristics of SOAs (e.g., loose coupling, published interfaces, standard communication model) offer the promise of enabling existing legacy systems to expose their functionality, presumably without making significant changes to the legacy systems[4]. However, constructing services from existing systems in order to obtain the benefits of an SOA is neither easy nor automatic. In fact, such a migration can represent a complex engineering task, particularly when the services are expected to execute within a tightly constrained environment.

SOA migration tasks can be considered from a number of perspectives including that of the end client or user of the services, the SOA architect, or the service provider. This paper focuses on the service provider.

2. Creation of Services From Legacy Components

Enabling a legacy system to interact within a service-oriented architecture, such as a Web services architecture, is sometimes relatively straightforward—this is a primary attraction to the approach for many businesses. However, characteristics of legacy systems, such as age, language, and architecture, as well as of the target SOA can complicate the task. An analysis needs to be performed to consider:

1. requirements from potential service users. It is important to know what applications would use the services and how they would be used. For example, what is the information expected to be exchanged? In what format?
2. Technical characteristics of the target environment, such as bindings, messaging technologies, communication protocols, service description languages, and service discovery mechanisms.
3. The architecture of the legacy system., including dependencies on commercial products or specific operating systems, or poor separation of concerns.
4. The effort involved in writing the service interface
5. The effort involved in the translation of data types.
6. The effort required to describe the services including information about qualities of service, such as performance, reliability, and security; or service level agreements (SLAs) .
7. The effort involved in writing service initialization code and operational procedures.
8. Estimates of cost, difficulty, and risk.

To gather this information and identify the risks for the migration effort in a systematic way, we have developed the Service-Oriented Migration and Reuse Technique (SMART). SMART is based on the OAR [1]method for

evaluating the reuse potential of legacy components, but customized to reflect the migration of components to services. Its activities are:

1. Establish stakeholder context
2. Describe existing capabilities
3. Describe the future service-based state
4. Analyze the gap between service-based state and existing capabilities
5. Develop strategy for service migration

These five activities are briefly outlined below.

Establish Stakeholder Context

In order to establish the context in which the migration to services will take place, SMART first identifies the stakeholders, including the current end users of the legacy systems, the potential end users of the migrated service operating within the SOA, and the owners of the legacy systems. The activity identifies who knows most about the legacy system, what it currently does, and what it should do as a service or set of services.

Describe Existing Capabilities

The goal of the second activity is to obtain descriptive data about the legacy components. Basic data solicited includes the name, function, size, language, operating platform, and age of the legacy components. Technical personnel are questioned about the architecture, design paradigms, code complexity, level of documentation, module coupling, interfaces for systems and users, and dependencies on other components and commercial products.

Historical cost data for development and maintenance tasks is collected to support effort and cost estimates.

Describe the Future Service-Based State

The two goals of the third activity are to:

- Gather evidence about potential services that can be created from the legacy components
- Gather sufficient detail about the target SOA to support decisions about what services may be appropriate and how they will interact with the architecture

Initial information about potential services often comes via conversations about the function(s) of the legacy system during the second activity. However, the information gathered often must be tempered by data from users, corporate architects, domain groups, communities of interest, and reference models that address service definition. In some cases, these groups and models will define the entire set of services that support the organization's goals, and into which any potential services built from the legacy components must fit.

Analyze the Gap

The goal of the fourth activity is to identify the gap between the existing state and the future state and determine the level of effort needed to convert the legacy components into services. This analysis may also suggest potential tradeoffs between the target architecture and the legacy components.

The tasks of this activity include:

- Develop an analysis strategy for legacy components that are being considered for migration.
- Analyze the legacy components to determine the types of changes that need to be made to enable migration. SMART uses three sources of information to support the analysis activity. The issues, problems, and other concerns that were noted as the team completed the previous, discovery-oriented steps form one source of information. A second source of information is provided by a Service Migration Inventory (SMI) that distills the many desired traits of services executing within SOAs into a set of topics. The team uses the SMI to assure broad coverage and consistent analysis of difficulty, risk, and cost issues. A third, optional source of information involves the use of code analysis and architecture reconstruction tools to analyze the existing source code for legacy components.

Develop Strategy for Service Migration

A key feature of SMART involves building cost projections for each migration option still under consideration. This is accomplished by considering organizational characteristics, difficulty and risk associated with various migration options, and applying historical productivity numbers where possible.

3. Pilot Application of the Process

An early version of SMART was applied in a recent pilot analysis of the potential for migrating a set of legacy components from a DoD command and control (C2) system to an SOA.

Establish Stakeholder Context

We initially met with the government owners of the system and the contractors who had developed the system. At this meeting we were given an overview of the set of systems, the history of the systems, the migration plans, and the drivers for the migration. We were given a brief orientation to the SOA and were also provided with system documentation.

The owners of the systems recognized that if a selected set of components from their C2 system are converted to application domain services within a specific target SOA, they may have applicability for a broad variety of purposes. Our role was to perform a preliminary

evaluation of the feasibility of converting a set of their components to application domain services within a SOA.

Describe Existing Capabilities

The pilot C2 system has two parts: 1) a mission planning system and 2) a mission execution system that adds situational awareness to the planning capability. These two systems were initially developed as part of a product line. Both rely on a set of core components for the data model, data analysis, and visualization.

Given the information about the target SOA, we met with the contractor and representatives of the government to focus on a limited number of legacy components and to select criteria for further screening. We focused on seven potential services that the government team had previously identified as part of its initial analysis of ADS requirements. These seven potential services contained 29 classes.

The current system, written in C++ on a Windows operating system, had a total of about 800,000 lines of code and 2500 classes. In addition, the system had dependencies on a commercial database and a second product for visualizing, creating, and managing maps. Both commercial products have only Windows versions.

The 29 classes that we selected enabled us to focus on potentials for high payoff. In conjunction with the team, we developed criteria for screening the potential reusable components. These criteria included:

- Size
- Complexity
- Level of documentation
- Coupling
- Cohesion
- Number of base classes
- Programming standards compliance
- Black box vs. white box suitability
- Scale of changes required
- Commercial mapping software dependency
- Microsoft dependency
- Support software required

These criteria formed the basis for the more detailed analysis discussed below.

Describe the Future Service-Based State

The system owner had done a preliminary identification of potential services that could be built from components of the legacy system. This analysis was derived from high level requirements for applications that were being targeted as users of services to be provided by the SOA. The system owner had matched legacy functionality to these high level requirements and provided some initial estimates of the contents of the potential services.

We investigated the target SOA through an analysis of available documentation and through a meeting with the developers. Because the SOA was still under development, the specifications for how to deploy and write services were still unclear.

Analyze the Gap

Given the known and projected constraints of the target SOA, we performed three different types of analyses: 1) an analysis of the changes to the legacy components that would be necessary for migration to the SOA, 2) an informal evaluation of code quality, 3) an architecture reconstruction to obtain a better understanding of the set of undocumented dependencies. The results of these analyses allowed us to define a service migration strategy based on the risks due to the unknown future state of the target SOA..

Analysis of Required Changes

We initially met with the contractor to get an understanding of the required changes, as well as estimates of the level of difficulty and the risks of making the changes. The contractor provided estimates for converting the components into services, based on a set of simplifying assumptions on the actual make-up of the target SOA and the final set of user requirements. These estimates initially suggested that the level of difficulty of making these changes would be low to medium, and the risk would be low because of their familiarity with the systems.

However, we found that the tools in use on the project only picked up first-level dependencies between classes. This indicated that the coupling and the amount of code that was used by each class was higher than could be estimated from the existing documentation. There was also no consistent programming standard, leading to idiosyncrasies between different programmers. Because of the inadequacies that we found in the architecture documentation, and the underestimation of the amount of code used by the potential services, there remained a number of gaps in our understanding of the system.

Code Analysis

To address remaining issues, we first analyzed the code through a code analyzer *"Understand for C++"*.

The code analysis enabled us to validate the input from the contractor and to produce input for the architecture reconstruction tool that would identify dependencies.

From the code analysis, we found that the code was better organized and documented at the code level than most code that we have seen. However, there were inconsistencies in the quality and documentation between different parts of the code that made the analysis complicated.

Architecture Reconstruction

To address the issue of dependencies in more detail, we conducted an architecture reconstruction with a tool called *ARMIN*. Architecture reconstruction is the process by which the architecture of an implemented system is obtained from the existing system [3].

In our analysis, we were interested in

- Dependencies between services and user interface classes
- Dependencies between services and the commercial mapping software
- Dependencies between services
- Dependencies between the services and the rest of the code that mainly represented the data model

The architecture reconstruction was able to identify a substantial number of undocumented dependencies between classes. These will enable a more realistic understanding of the scope of the migration effort if it succeeds.

The architecture reconstruction also enabled us to document the central role of the data model, and to identify it as a potentially valuable reusable component, even though it had not been identified during the initial analysis.

Develop Strategy for Service Migration

In looking at the potential for reuse of the existing legacy components, we found that the current legacy code represents a set of components with significant reuse potential. However, because the current legacy system does not have sufficient architecture or other high level documentation, it was difficult to understand the "big picture" as well as dependencies between different classes. The largest risk in reusing the legacy components concerns the fact that the SOA has not been fully developed. We also recommended that the government organization require the following changes from its contractors to make reuse of its legacy components more viable:

- Suitable set of architectural views
- Consistent use of programming standards
- Documentation of code to enable comments to be extracted using an automated tool
- Documentation of dependencies, especially when they violate architecture paradigms

4. Conclusions and Next Steps

We found that the initial task of determining how to expose functionality as services, while seemingly straightforward, can have substantial complexity. Our conclusions to the client, while not definitive, did point out a number of issues that they had not previously considered. The type of disciplined analysis that we performed appears to have applicability for other organizations that are considering migrations to SOAs.

References

[1] Bergey, J.; O'Brien, L.; and Smith, D. "Using the Options Analysis for Reengineering (OAR) Method for Mining Components for a Product Line," 316-327. Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2). San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002.

[2] Brown, A; Johnston, S.; and Kelly, K. Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications. Rational Software Corporation. 2002.

[3] Kazman, R; O'Brien, L.; and Verhoef, C. Architecture Reconstruction Guidelines, 2nd Edition (CMU/SEI-2002-TR-034). Software Engineering Institute. November 2003.

[4] Lewis, Grace and Wrage, Lutz. Approaches to Constructive Interoperability (CMU/SEI-2004-TR-020). Software Engineering Institute. January 2005.

Interoperability and Integration of Enterprise Applications through Grammar-Based Model Synchronization

Igor Ivkovic and Kostas Kontogiannis
Dept. of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L3G1 Canada
{iivkovic, kostas}@swen.uwaterloo.ca

Abstract

Enterprise applications consist of heterogeneous components and subsystems that communicate through externally exposed interfaces. Each of the component or system interfaces represents a set of specific constraints and relations to which the systems that implement them must conform. Therefore, the problem of enterprise application integration and interoperability can be seen as a problem of model synchronization of system descriptions and the underlying deployment component models. In this paper, we present an approach to integration and interoperability of enterprise applications through grammar-based model synchronization. As part of the approach, component and subsystem interfaces are viewed as instances of corresponding domains formally defined through annotated domain model grammars. The automated mapping between source and target interfaces is performed via association grammars that represent associated productions of the two domains.

1. Introduction

As dependence of business models on technology grows and becomes more complex, the need for integration of heterogeneous applications into unified business processes is growing in priority. Development of individual software systems is now coupled with specification and implementation of interoperability and integration solutions, with an implicit technical requirement that systems must bind together into encompassing enterprise applications to satisfy a range of functional and nonfunctional business requirements.

The terms interoperability and integration in the context of Enterprise Application Integration (EAI) are used interchangeably but they carry a different meaning [4]. Interoperability approaches are those that focus on the exchange of data and control information between independent software systems. On the other hand, integration approaches aim to unify events and messages used by the including systems into a single enterprise architecture, where different applications are a part of one large logical unit. As a common point, both interoperability and integration focus on interface technologies that provide entry and exit points for the source and target in communication between different enterprise components and subsystems.

Different types of interface technologies can be identified including proxies, connectors, agents, adapters, and many others. The distinguishing characteristics include (1) adaptation, which concerns capability to translate and accept different types of incoming or outgoing data, (2) simulation, which concerns capability to expose component functionality and related application programming interfaces (APIs), (3) security, which concerns means to protect the incoming and outgoing data transports, (4) authentication, which concerns means to confirm the identity of source and target components, *etc*. [3]

Under the pressure of expanding diversity of components and interfaces used in creation of enterprise applications and the need to decrease the length of change cycles that include integration-related development, automation of suitable aspects of EAI is becoming a necessity [1]. In this paper, we propose an approach to enterprise integration which focuses on automating suitable aspects of integration of component and subsystem interfaces through grammar-based model synchronization. As part of the approach, components and subsystem interfaces and their functionality is first represented in terms of types and relations specified through corresponding domain models and formalized using annotated domain model grammars. The annotations of grammar productions are with respect to meta information and constraints that pertain to specific types of interfaces (*e.g.*, adaptation-capability = none). Based on the mapping of encoded semantic annotations, the source and

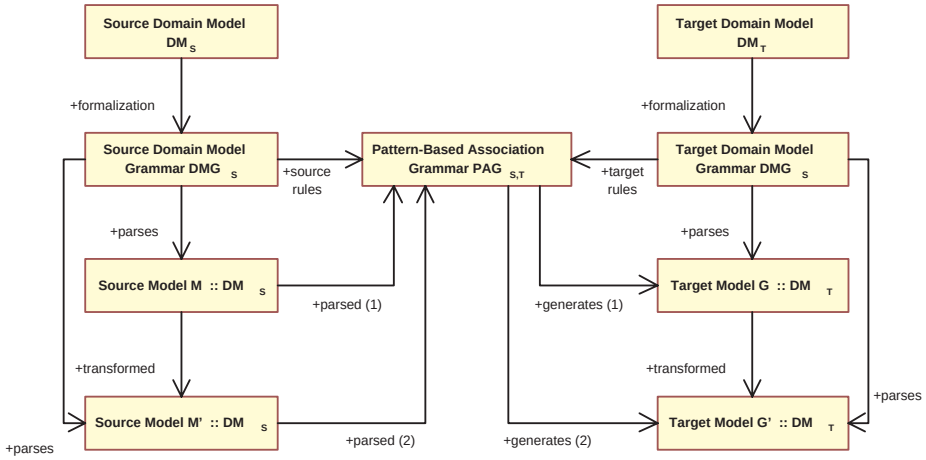


Figure 1. Grammar-Based Model Synchronization

target productions are combined into association grammars. Finally, using the association grammars, one can automatically generate the target interface model from the source interface model that can be used to address a specific aspect of enteprise integration.

The rest of the paper content is organized as follows: Section 2 introduces grammar-based model synchronization. Section 3 presents the application of the methodology to the domain of EAI, specifically synchronization of component and subsystem interface with regards to heterogeneous data integration, while Section 4 gives our conclusions and directions for future research.

2. Grammar-Based Model Synchronization

In model-driven software evolution, a concrete model, besides conforming to its metamodel, also relates to an application domain context (*e.g.*, web development, content management). For each domain context, specific types and relations apply. The domain context is formally denoted as a domain model, which may be created as part of the design documentation or later extracted using a domain analysis technique such as FODA [2]. Additional domain-specific meta information that include ontologies and feature maps may be denoted using domain-model attributes. With formally defined domain models at hand, concrete models can be represented in terms of domain-specific concepts and annotated with domain-specific meta information to become more lucid and in turn better support development activities for which they were intended.

In grammar-based model synchronization, domain models are represented as semantically-annotated context-free grammars to which we refer as domain-model grammars. The types and relations contained within a domain model are represented as grammar production rules with additional meta information encoded as semantic heads preceding each rule. The emphasis of the proposed approach is on consistency management of heterogeneous artifact models, so we consider associations between model elements of different domains and encode dependencies as associations between the source and target grammar rules based on the matching of their semantic heads. The association rules between source and target productions are formally represented as pattern-based association grammars. Starting with the source model, using the association grammar, one can automatically generate a target model that is synchronized with the source. The generated target model can be used in the further course of model-driven software evolution.

Figure 1 illustrates the proposed approach. In specific terms, the goal of the framework is to generate from a source model M instantiated from a domain model DM_S a corresponding target model G that conforms to a domain model DM_T. The domain models DM_S and DM_T are denoted as domain-model grammars DMG_S and DMG_T respectively. The production rules from the two grammars are used to define a pattern-based association grammar [5], $\mathrm{PAG}_{S,T}$, that provides a mapping between source and target production rules. The association rules of $\mathrm{PAG}_{S,T}$ are created through the matching of semantic heads of source and target domain model grammars. Using the association grammar $\mathrm{PAG}_{S,T}$, the source model M is parsed and the source rules used in the derivation tree T_M are mapped to corresponding target rules to create a target derivation tree T_G. The target model G is generated from the target tree T_G and it can be used directly in the following evolution activities or further transformed to G'.

To arrive at the proposed solution to the problem of model synchronization, we apply already established the-

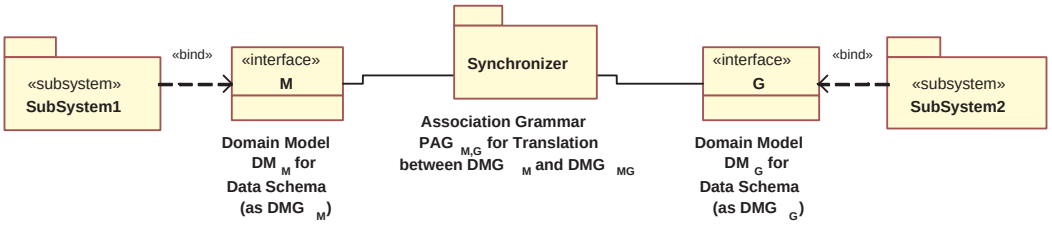


Figure 2. Enterprise Data Integration through Model Synchronization

ory from the area of natural language processing (NLP) to a more structured domain of model-driven software evolution. NLP theory provides frameworks for (1) representing models as sentences that comply to corresponding grammars, (2) encoding of semantic properties as semantic heads, (3) association of source and target grammars using pattern-based association grammars, and (4) automatic translation by automatic mapping of source and target model parse trees.

3. Enterprise Data Integration through Grammar-Based Model Synchronization

To apply the proposed framework to the domain of EAI, we focus on component and subsystem interfaces. We represent integration as synchronization of the interfaces, that is, their data schemas represented as specific domain models. To enable synchronization, we introduce a *Synchronizer* component in the enterprise architectures that is capable of translating data streams between two component and subsystem interfaces. Any input stream that complies with a specific input data schema is automatically mapped to an output stream that complies with a specific output data schema. Both input and output schemas are denoted as domain models and mapped through the Synchronizer component using grammar-based model synchronization. The aim of the approach is to provide a framework for automating data integration where the users can tailor specific translation mechanism to their own needs (*e.g.*, through selection and encoding of semantic heads).

The steps in the application of this approach are as follows (as illustrated in Figure 2):

1. Represent data schemas of interfaces M and G as domain model grammars DM_M and DM_G by encoding domain-specific types and relations.

2. Automatically denote DM_M and DM_G as domain model grammars DMG_M and DMG_G and annotate, as necessary, the grammars with properties regarding specific constraints and meta information (*e.g.*, ontologies, features maps).

3. Match the source and target productions to derive the corresponding association grammar $\text{PAG}_{M,G}$ for translation of data from M to G (or encode $\text{PAG}_{G,M}$ for translation of data from G to M).

4. Use $\text{PAG}_{M,G}$ to automatically perform the mapping of data from M to G.

As a demonstrative example, please consider the following two data schemas:

- M that contains types Staff, Address, and Employer; and
- G that contains Employee, Residence, and Company.

The corresponding domain models grammars for M and G are as follows.

DMG_M:

M → Staff | Staff M | Address | Address M | Employer | Employer M
[Ontology-Match=Person] Staff → FirstName LastName
[Ontology-Match=Location] Address → AddressLine1 AddressLine2 City PostalCode Country
[Ontology-Match=Organization] Employer → EmployerName EmployerType

DMG_G:

G → Employee | Employee G | Residence | Residence G | Company | Company G
[Ontology-Match=Person] Employee → FirstName LastName Title
[Ontology-Match=Location] Residence → AddressLine1 AddressLine2 City PostalCode Country
[Ontology-Match=Organization] Company → CopmanyName

A corresponding association grammar for DMG_M and DMG_G is as follows. The grammar is created using meta information (*i.e.*, a simple integration ontology) as the basis for matching of the source and target grammar rules.

PAG$_{M,G}$:

[Ontology-Match=Person] (Staff → FirstName LastName) → (Employee → FirstName LastName Title)
[Ontology-Match=Location] (Address → AddressLine1 AddressLine2 City PostalCode Country) → (Residence → AddressLine1 AddressLine2 City PostalCode Country)
[Ontology-Match=Organization] (Employer → EmployerName EmployerType) → (Company → CopmanyName)

Using PAG$_{M,G}$, it is now possible at run-time to automatically map the input stream coming as output of interface M to an output stream as the input for interface G.

4. Conclusions

In this short paper, we have presented an approach to integration and interoperability in enterprise application integration using grammar-based model synchronization. We have discussed the basics of our approach and showed how the approach can be used on one of the aspects of enterprise integration, namely, data integration.

In future research, we intend to describe the application of our framework in more detail and extend the capacity of the approach to deal with other problem scenarios of enterprise integration such as application integration.

References

[1] D. A. Fisher and D. Smith. Emergent issues in interoperability. *SEI Eye on Integration*, 3, 2004.

[2] L. Kean. Feature-oriented domain analysis. Software technology review, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1997.

[3] D. S. Linthicum. *Enterprise Application Integration*. Addison-Wesley, Reading, MA, 2000.

[4] J. T. Pollock. The big issue: Interoperability vs. integration. *eAI Journal*, Oct 2001.

[5] K. Takeda. Pattern-based context-free grammars for machine translation. In *Proceedings of the 34th conference on Association for Computational Linguistics*, Santa Cruz, CA, Jun 1996.

Interoperability for Data and Knowledge in Healthcare Systems

Kamran Sartipi and Reza Sherafat
Dept. Computing and Software
McMaster University
Hamilton, ON. L8S 4K1, Canada.
{*sartipi, sherafr*}*@mcmaster.ca*

Kostas Kontogiannis
Dept. Electrical & Computer Engineering
University of Waterloo
Waterloo, ON. N2L 3G1, Canada
kostas@swen.uwaterloo.ca

Abstract

In this paper we propose a reference architecture for distributed healthcare systems consisting of intelligent databases that contain administrative and/or clinical patient data within a healthcare institution such as a hospital, a healthcare provider, a laboratory, or a governmental organization. The reference architecture takes advantage of new models and enabling technologies and supports active participation of the healthcare personnel. The context-aware knowledge acquisition mechanism uses agent technology to provide additional information with respect to the patterns and trends in knowledge for decision making purposes. The resulting healthcare system would assure enhanced level of clinical diagnosis, reduced cost in health services, better planning and scheduling for service fees, and ease of use and flexibility of features. Integration of data mining models with the standard HL7 data models allows the interoperability among heterogeneous healthcare systems at a higher level of knowledge mining. In such an environment the healthcare personnel obtain proper assistance in a secure and privacy-assured service oriented distributed healthcare environment.

1 Introduction

In this paper, we take the position that by incorporating the knowledge management and data mining models into the standard healthcare environments we can achieve some level of knowledge interoperability to support the helathcare personnel in administrative and clinical decision making purposes. The huge size of existing patient data; transition from paper-based patient records to electronic files; and mutual agreements and understandings between physicians and IT experts in better handling the sensitive patient data, are all evidences of entering to a new era of healthcare informatics that mostly rely on IT technology. The intrinsic heterogeneity of the healthcare systems and the emerging demands in providing efficient, reliable, on-demand, and knowledge-based information to the healthcare personnel necessitate a high-level of data and service interoperability among such systems. For today's healthcare environments, the access to clinical and administrative decision support techniques is critical. Despite much attention has been paid towards provision of data interoperability for healthcare system, knowledge management and in particular discovery of patterns and trends in knowledge are not much supported by healthcare standards. This knowledge when combined with data interoperability standards can: enhance the efficiency and quality of healthcare services; save huge amount of costs related to data inconsistency, duplication, and human errors; and promote the accuracy of patient diagnosis through additional knowledge provided by the information system. In recent years, growing shares of the healthcare organizations' revenues are spent on IT and there is also promising developments in brining all activities under the same umbrella through standardization which provides better communications and interoperability. HL7 (Healthcare Level 7) [1] standard messages defined over a standard Reference Information Model (RIM) cover much of the data modeling needs of the healthcare.

In brief, the goal of this research is to provide a flexible and customizable reference architecture for the healthcare domain, by focusing on incorporating knowledge management underlying platform with the standard data interchange models. More specifically, the significance of the proposed research on healthcare architectures is evinced in: i) evaluating information system integration standards and technologies in the context of distributed and interoperable healthcare systems; ii) providing the mechanisms for knowledge management service integration with the underlying platform to support administrative and clinical decision making; iii) providing an enhanced user-system interaction environment for knowledge acquisition that is characterized by: a high-level language, agent-based technology, and context-aware knowledge management techniques; and iv) incorporating the required data model to achieve flexibility and ease in implementing security and privacy policies.

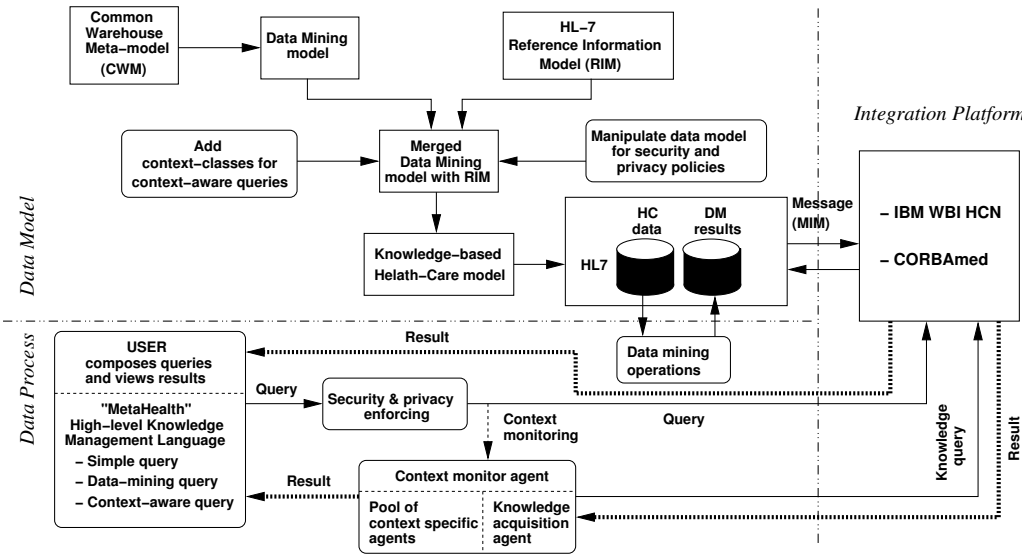


Figure 1. The proposed reference architecture for emerging demands in healthcare systems.

2 Proposed healthcare reference architecture

The proposed reference architecture for the healthcare domain is illustrated in Figure 1. Based on the particular features of a specific healthcare domain the reference architecture can be specialized to support different types of knowledge acquisition, security, and privacy techniques while complying with the international data model standards. The architecture consists of three parts that are discussed below.

Data Model

The data model has been designed to comply with the internationally accepted standards for healthcare domain HL7 RIM (Health-Level 7 Reference Information Model) and knowledge management domain CWM (Common Warehouse Meta-models) [2]. Based on the objective of the analysis the RIM model can be considered as a group of incorporating and overlapping class diagram segments, where each segment (a subject area such as administrative, diagnostic, or clinical) represents the interest of a Technical Committee (TC) that work on the healthcare information. We would study the ways to augment the HL7 RIM object model with a data mining sub-model that would allow us to perform data mining operations. This can be viewed as adding a new knowledge-management subject area to the RIM. In order to do this, we use the Common Warehouse Meta-models that defines the relations among the types of classes and their relationships that support different data mining algorithms, and would fuse those classes and their relations within the RIM model. The reference architecture would allow to add classes that represent the context of the queries issued by the user in order to provide a context-aware query generation. Similarly, the data model should be manipulated to allow implementation of the security and privacy polices that would provide a support for the corresponding processes in the "process model" that would be described below. The resulting data model would serve as a knowledge-based data transfer model that would support communication between healthcare applications in the proposed healthcare reference architecture. The instantiation of this joint data model and populating it with the corresponding data can be represented as two databases: a healthcare database from the HL7 RIM model (HC), and a data mining database (DM) that contains the results of the data mining algorithms on the healthcare database. In this way, we can define different privacy and security policies for each database and also define different performance and maintenance requirements for each. In this architecture, the healthcare applications would communicate using HL7 standard for communication. A typical healthcare scenario (or event) such as "*admitting patient X in hospital A and retrieving his/her file from hospital B*" would be decomposed into a number of HL7 messages using the HL7 MIM (Message Information Model) to be transferred between the healthcare databases in hospitals A and B. However, a knowledge-based scenario such as "*obtaining the geographically-spread patterns of the patient X disease*" is decomposed into messages that would be communicated between the data mining databases in hospital A and another hospital (may be B or C).

Data Process

The data process specifies the kind of processes that operate on the original and mined healthcare data in the corresponding databases. The core of this part is a high-level

language “MetaHealth” that supports both the knowledge management and data retrieval requirements of the system in three categories of queries, as follows:

1) Basic level: the MetaHealth language provides data retrieval and manipulation capabilities in the same manner that SQL does. 2) Knowledge mining level: the language provides ways to explicitly mine the healthcare database for new knowledge that the user’s application might request. This knowledge might either be available, i.e., due to previous computations and performing a quick computation (as in case of light weight agents), or can be computed from scratch. 3) Context-aware level: at the highest level, the knowledge access and retrieval becomes an underlying and insensible processing of the data retrieval request. As for the basic data access, the user query is processed but in addition to that, the system puts some additional information when sending the data. This information is subject to explicit server-level instructions that can be defined at the server. For example the server might be configured to retrieve any new patterns that had been discovered in advance for each requested field. This helps the user to have access to the available knowledge seamlessly. Furthermore the retrieved knowledge should be trimmed on the context in which the user performs the query. The information that is added in the reply should be prioritized by the significance of the knowledge. For instance if due to the outbreak of a disease in recent months, the risk of a certain age group is increased the system can put this caution before less risky diseases. Also the user should not be overwhelmed by too many messages and warnings. Alternatively, we can provide client side mechanisms that trim and filter the excessive information based on user preferences. The context within which the user accesses the data is also important. A physician or a pharmacist can have different perspective at different situations. Each healthcare service that represents a scenario (event) will be published to a server and would be available for the subscribing systems.

The proposed approach will take advantage of the software agent technology to achieve a novel and sophisticated context-aware query mechanism. This allows autonomic behavior for the proposed architecture which is able to react upon receiving a request for accessing or manipulating data. In this context, a software agent, namely “context monitor agent”, will monitor the specific fields of the queries issued to a database in order to pick context-based information which will be used to select a “context specific agent” from a pool and, pass the context information for further process. Each context specific agent has a different mission and will provide exploratory and organized knowledge to the expert user by issuing further queries to a database to obtain required information, processing the obtained information from database, and properly presenting the results to the user for further interpretation.

Both the security and privacy requirements can be enforced by the means of techniques that would define hierarchies of privileges for different roles in a healthcare system. We will study the techniques bases on privacy preferences such as Hippocratic database [3] that can be represented as classes in the data model and represented in the database; these preferences would be checked against the organization’s privacy policy to either allow or block the query. The security provision would be achieved by defining a hierarchy of security concerns that allow from low-level and concrete security regulations up to the high-level and abstract ones. We intend to adopt internationally consensused standard techniques such Common Criteria [4].

Integration platform

The discussed reference architecture constitutes the application layer of the healthcare distributed system which must be supported by a standard integration platform that provides the required service based communications means. The standard service-based platforms such as IBM WBI (WebSphere for Business Integration) technology provides the required middleware support with a wealth of features that cover the HL7 standards for interoperability, user mobility, distributed communication, and standard security provisions. The IBM WDI (WebSphere Data Interchange) supports HL7 standards for data transformation that allows the data standards of the healthcare organizations to be communicated with HL7 message standards. The proposed architecture will aim at shifting some of the security and privacy features of the healthcare system to be implemented in the middleware of the adopted platform. Another alternative platform would be CORBA technology.

Challenges

The challenging issues in the proposed model follows. Securing large and reliable health databases to evaluate the effectiveness and usefulness of a prototype healthcare system; our research team expect to access to a several up-to-date large health databases. ii) Active collaboration of the healthcare personnel (medical researcher, healthcare organizations) with the research team to provide a proper testing environment. We are working with healthcare providers that will allow us to test the system in the site. iii) Assuring the privacy of the sensitive patient data.

References:

[1] http://hl7.org
[2] http://www.omg.org/cwm/
[3] http://citeseer.ist.psu.edu/
agrawal02hippocratic.html
[4] http://www.commoncriteriaportal.org/

Policy-Based Orchestration of Autonomic Elements

Mazeiar Salehie, Ladan Tahvildari
Dept. of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
{msalehie, ltahvild}@uwaterloo.ca

Abstract

Autonomic systems encompass of components, applications and systems. Composition, integration and orchestration at different levels of such autonomic systems are significant issues in their design, evolution and operation. This research focuses on orchestration in a simulated autonomic system. Policy-based management has potential to play an important role in orchestrating constituent components of autonomic systems (autonomic elements) more effectively and also filling the gap between business and IT objectives. We use policy-based orchestration, by crisp and fuzzy policies, to reach the business goals of the system and to obey the constraints defined by the Service Level Agreement (SLA).

1 Introduction

In order to cope with the ever increasing complexity of managing distributed, heterogeneous computing systems, it has been proposed that systems themselves should manage their own behavior in accordance with high-level objectives and policies [9]. Such autonomic systems consist of autonomic elements (components), which each one in turn encompasses of an autonomic manager and a managed resource(s). The interoperability of these constituent components has a vital role in effectiveness and efficiency of the entire system [11]. In fact, to satisfy the objectives of the system, autonomic elements need to be composed/integrated (for collaboration) and/or be orchestrated (for coordination). So, in a higher level, there are also components for orchestration and dynamic composition (of components, services and resources) similar to what defined in IBM architectural blueprint [1].

Similar to component-based systems, two of major concerns in designing autonomic systems are [8]: i) composition of components, choreography and orchestration, and ii) Quality of Service (QoS) supporting infrastructure. These two issues are important at both global and local levels of a system. Litoiu et al. [7] propose a hierarchical model for controlling the system regarding performance and QoS at three levels namely: component, application and system. They defined three operations of tuning, load balancing and provisioning respectively from component to system level and used the idea of QoS-aware components for self-optimizing. The important point in such a view is the policy propagation from the system level (highly related to SLA) to lower levels. The problem is how to reconcile global coordination with local control/decision making. The main focus of this paper is on the policy-based orchestration of autonomous elements to solve this problem. Policies are related to business policies, SLA and QoS in different levels of the system.

The remainder of this paper is organized as follows. Section 2 reviews fundamental concepts of orchestration, policy and policy-based orchestration. Section 3 presents the simulation model used for the experiments, and discusses the results obtained by applying policy-based management method on the model. Finally, Section 4 summarizes the contributions of this work and outlines directions for further research.

2 Policy-Based Orchestration

One of the current hot topics in autonomic computing is orchestration of elements, applications or systems in large systems like data centers. Orchestration is co-related to a sort of optimization both as the initial optimization of elements, and keeping these elements optimized when changes occur. Current goals for such optimizations typically focus on IT measures (e.g. increasing the system's availability). Nowadays, however, many enterprises are interested in business objectives, such as maximizing the total income. Therefore, IT optimization should focus on these business goals, rather than the more conventional technical metrics.

Systems such as Oceano [4] use a SLA-based management, which extracts metrics from SLA to define the constraints and high-level directives, policies of the system.

Optimizing the IT infrastructure according to such business objectives is not a trivial task, as it is unclear how configurations at the IT level will impact the business objectives. Policy-based management is a paradigm to solve this problem. Policies could be used for both integration/composition and orchestration/choreography of components and services. First of all, let us define the fundamental concepts such as orchestration and policy.

2.1 Orchestration

The term *Orchestration* is generally means[1]: "an arrangement of events that attempts to achieve a maximum effect". So, Orchestration is a kind of coordination and mostly it is compared with choreography. The difference is that the former more deals with coordination while the latter is about collaboration between two or more systems.

A single autonomic manager acting in isolation can achieve autonomic behavior only for its managed resources. The autonomic managers need to be coordinated to deliver system wide autonomic computing behavior. Orchestrating autonomic managers provide this coordination function. There are two common configurations [1]: 1) Orchestrating within a discipline which coordinates multiple autonomic managers of the same type (one of the self-CHOP - configuring, healing, optimizing and protecting). 2) Orchestrating across disciplines which coordinates autonomic managers that are a mixture of self-CHOP. An example of an orchestrating autonomic manager is a workload manager. An autonomic management system for workload might include self-optimizing autonomic managers for particular resources, as well as orchestrating autonomic managers that manage pools of resources.

IBM has proposed a 5-layer reference architecture for autonomic systems [1]. The lowest layer contains the managed resources (hardware or software) and the next layer incorporates consistent, standard manageability interfaces for accessing and controlling the managed resources through a touchpoint. Layer three illustrates one autonomic manager for each of the four self-CHOP and layer four contains autonomic managers that orchestrate other managers. The top layer provides a common system management interface for the IT professional through an integrated solutions console.

2.2 Policy

Numerous and variant definitions of policy (e.g. [3, 10], to name a few) have been proposed in recent years. Most of them noted that policies are a form of guidance used to determine decisions and actions. Any single definition of policy is likely to be too restrictive to cover all of the variant forms of behavioral guidance that will be needed. Instead, in autonomic computing community, researchers choose a very broad definition of policy [5]: "a policy is any type of formal behavioral guide". Clearly, then, policies have a fundamentally significant role to play in an autonomic system.

One noteworthy point is that policies should cover the entire state space and provide unambiguous guidance to the system. Because this cannot generally be achieved by considering individual policies in isolation, policy sets are the right way to think about policy. Unfortunately, people typically think of policies individually, because understanding and reasoning about policy sets in real complex systems are difficult.

Kephart et al. [5] defined three types of policies:1) Action Policies which dictate the action that should be taken whenever the system is in a determined current state and typically this takes the form of condition-action (If-Then), 2)Goal Policies which rather than specifying exactly what to do in the current state, specify either a single desired state, or an entire set of desired states, and 3) Utility Function Policies which are objective functions that express the value of each possible state. Utility Function policies generalize goal policies. Instead of performing a binary classification into desirable vs. undesirable states, they ascribe a real-valued scalar desirability to each state. Utility functions are well-known in the fields of economics and artificial intelligence as a form of preference specification [12].

Some other researchers like Simmons et al. [10] provided a longer list of different policy types namely: service-level, system-level, configuration, resource conflict, authorization and cost-estimation for not satisfying a resource demand. This paper focuses on action and utility policies in an autonomic system, because at least realizing these types as a policy set is more straightforward than the other complicated types of policies.

3 Simulation

For presenting the impact of policy-based management on orchestrating, we have built a simulation model similar to a simple data center. The orchestrator in this model uses "orchestration within a discipline", in which the discipline is self-optimizing. This autonomic system uses a solicited orchestration, which means autonomic elements request from orchestrator, and the orchestrator coordinate the elements using the global policies.

3.1 Simulation Model

The simulated model used in this paper is similar to the model in [12]. Figure 1 shows the simulated model, in which application managers control the local resources and the Global Autonomic Orchestrator (GAO) makes decision about the resource distribution and provisioning in the system level. In this abstract model application managers predict the workload and future demands of each application.

A rudiment definition for a managed resource is: "an entity that exists in the run-time environment of an IT sys-

[1] Wordnet definition

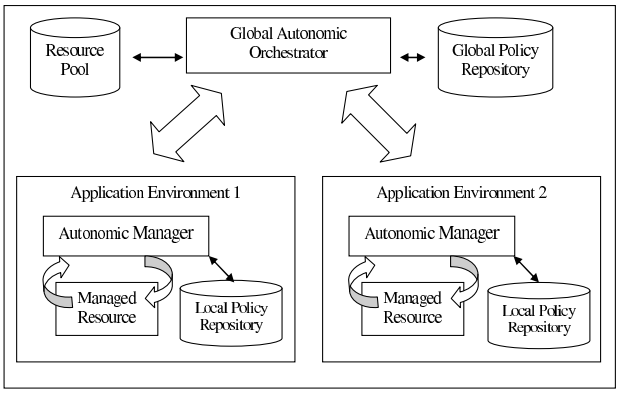


Figure 1. Simulation Model.

tem that can be managed." For this simulation, managed resources are servers (e.g. web servers) which by input requests, denoted as R, and total available CPU from servers, denoted as C, give the response time, denoted as RT. Business rules in this simulation define two types of service, gold and silver, and two different traffic sources has been considered for these two services. So we have two response times for two services: Total response time for each server is the summation of these two response times. The summation of C_G and C_S must always be less or equal of total CPU available (100 for each server). The autonomic manager in each application must adjust CPU shares to appropriate levels for each service based on predefined local policies. Kephart et al. in [5] have described how to use three types of policies (action, goal and utility) for the data center model. Walsh et al. in [12] have focused on utility-based policies for the data center model to maximize the total utility calculated by each application manager. This utility is calculated based on the current resource level, resource demand and predicted resource demand.

The application manager in the given model (Figure 1), does not use a prediction algorithm, but by checking the resource level threshold and the system performance (response time) predicts the resource demand. The interpretation of the utility function in this paper more relates to business objective and SLA as discussed in [13]. Actually, the main goal for utility-based policies is to maximize the business profit through not violating SLA for different services. The idea is similar to the work of Aiber et al. in [2]. They have defined business objectives and SLO (per day and per transaction benefit and performance penalty) and have tested a business-based self-optimizing method on a simulated system. The interesting part of their work is modelling the user behavior and load breakdown between servers in a multi-tier system (described in detail in [6].

3.2 Experiments

Table 1 shows the SLA used in the experiments on application controller. Each user gives a flat fee for daily service,

Service	Response Time	Flat Fee/Day/User	Response Time Penalty (%)
Gold	1 millisecond	50$	5$
Silver	2 millisecond	30$	3$

Table 1. SLA for Gold and Silver Services.

gold or silver, and the system calculates a penalty for violating the agreement on the response time. For some application a transaction-based fee can be also added (as described in [6]), which needs a more complex utility measurement. We have also defined two different business values for each application (0.8 for application 1 and 0.3 for application 2).

3.2.1 Experiments on the Application Controller

The main goal of the experiments on the application controller is to analyze the impact of policy-based management on performance of an application. For performance, the response time has been considered and two types of policy (action and utility) has been tested in the application level. In the experiments, it has been assumed that every 50 simulation steps is equal to one day and the default user traffic (silver and gold) in these experiments a ramp with slope 1 has been used due to the possibility of better testing the model especially in high competitive conditions. The model in some cases has been checked by a random generated traf-

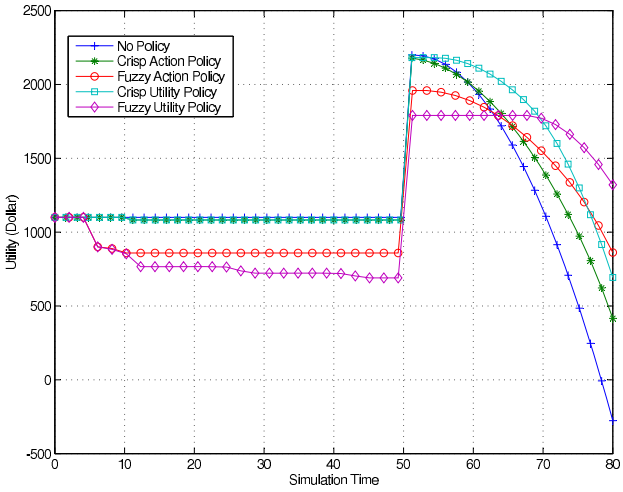


Figure 2. Utilities for the 1^{st} Experiment.

Figure 2 illustrates the utility of an application managed with different kinds of policies. Total utility has been improved by applying policies and utility policies had a better impact than action policies. Moreover, fuzzy policies show a better output utility due to the smoother decisions and having capability for better tuning by an administrator.

3.2.2 Experiment on Orchestration

This second experiment focuses on orchestration of applications in the system level. The orchestrator here is the GAO, which is in charge of resource provisioning in the system level. There is shared resource pool and GAO decides which applications are eligible to give resources. Sim-

ilar to previous set of experiments in the application level, GAO can use action or utility policies or even a combination of them. This experiments uses combination of action and utility policies for provisioning. Action policies have been used for non-competitive states (no conflicts on resources), while utility policies resolves the conflicts in the competitive states. We have used fuzzy utility policies due to the fact that these types of policies could better compromise between contestants in a conflict.

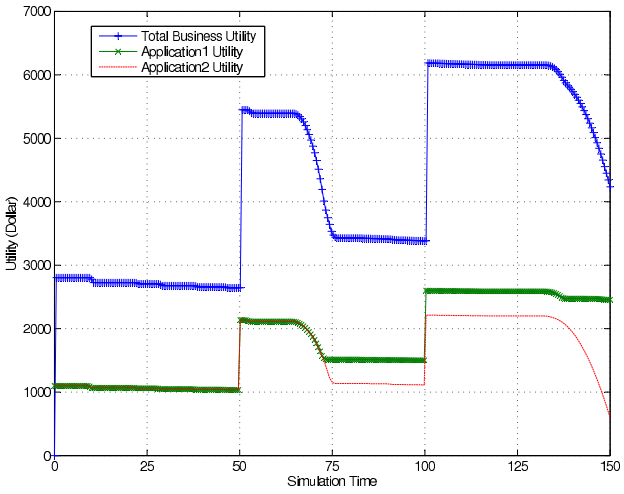


Figure 3. Utilities for the 2^{nd} Experiment.

Figure 3 illustrates the utilities for application 1, application 2 and total business utility, which is a weighted sum of the two applications' utilities. In this experiment, there are 3 servers in the global resource pool and each application starts its work with one server. As the output shows, GAO has assigned two of the servers to application 1 (at about time-step 74 and 140) and one server to application 2 (at about time step 75).

4 Conclusion and Future Works

The experiments accomplished in this paper, have shown that the policy-based management can help the controller, both application controller and GAO, to improve total business utility metrics and reaching the business objectives. Policy-based paradigm is not essentially a better mechanism for performance optimization (in terms of response time or throughput), but by externalizing the controlling knowledge of the system helps managers and administrators to better define policies, system objectives and high-level directives. By adding an appropriate solution for policy editing, analyzing and resolving conflicts, this mechanism can really improve quality of the management of a composed complex system.

Although the experiments done in this paper use a rather simple scenario for an autonomic system and its constituent autonomic elements, the added flexibility both in local and global levels of the system is remarkable. Because high-level policies are defined by human agents, administrators and managers, using fuzzy linguistic terms can help them to express policies more naturally. The experiments also showed fuzzy policy-based controllers behave considerably better than crisp controllers. For a reasonable number of rules and membership functions for input and output parameters, to be tunable by a human agent, fuzzy policies work better. Otherwise, machine learning techniques (such as neural network) should be used to tune the policies. An option that has not been tried in the experiments is priority for crisp and fuzzy policies. In the fuzzy context one option is to use certainty factor as priority for rules.

References

[1] An architectural blueprint for autonomic computing. IBM white paper, 2004. http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf.

[2] S. Aiber and et al. Autonomic self-optimization according to business objectives. In *Proc. of First International Conference on Autonomic Computing*, pages 206–213, 2004.

[3] N. Damianou and et al. The ponder policy specification language. In *Proc. of 2nd International Workshop on Policies for Distributed Systems and Networks*, 2001.

[4] G. Goldszmidt and et al. Oceano – sla based management of a computing utility. In *Proc. of 7th IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[5] J. O. Kephart and W. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proc. of 5th IEEE International Workshop on Policies for Distributed Systems and Networks*, 2004.

[6] A. Landau and et al. A methodological framework for business-oriented modeling of it infrastructure. In *Simulation Conference*, pages 464–472, 2004.

[7] M. Litoiu, M. Woodside, and T. Zheng. Hierarchical model-based autonomic control of software systems. In *Proc. of workshop on Design and evolution of autonomic application software*, pages 1–7, 2005.

[8] S. D. Panfilis and A. J. Berre. Open issues and concerns on component-based software engineering. In *WCOP'04*, 2004.

[9] M. Salehie and L. Tahvildari. Autonomic computing: emerging trends and open problems. *ACM SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[10] B. Simmons and H. Lutfiyya. Policies, grids and autonomic computing. *ACM SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[11] D. Smith, E. Morris, and D. Carney. Interoperability issues affecting autonomic computing. In *Proc. of workshop on Design and evolution of autonomic application software*, pages 1–3, 2005.

[12] W. Walsh and et al. Utility functions in autonomic systems. In *Proc. of First International Conference on Autonomic Computing*, pages 70–77, 2004.

[13] L. Zhang and D. Ardagna. Sla based profit optimization in autonomic computing systems. In *Proc. of 2nd international Conference on Service-Oriented Computing*, pages 173–182, 2004.

Integration of Perspectives and Issues of Different Types of Stakeholders in Service Oriented Architecture

Dennis Smith and Grace Lewis, Software Engineering Institute

Introduction

Organizations are adopting service-oriented architectures for a number of reasons, such as:

- To leverage the investment in legacy systems by providing a modern interface to existing capabilities
- To build systems around coarse-grained components (services) that can be used by many applications running on different platforms
- To leverage capabilities developed by external third parties

Although these are valid reasons and the amount of literature on the topic is large, there are no good published strategies for building end-to-end systems[1] based on SOAs; there are no approaches for understanding end-to-end quality of service, the technologies that SOAs are based on are still immature, and it is not clear what works and what does not work [Ma 05, Manes 05]. For example, SOAs have been emerging as a mechanism for achieving interoperability within and between systems.

Perspectives of Different Stakeholders

Despite the research on SOAs, most current work treats stakeholders as a single entity. Based on our work with large scale government systems, we have found it useful to distinguish between three types of components within an SOA, as presented in Figure 1:

- A set of applications that make use of services
- A set of services used by applications and other services—composed of service interfaces and underlying custom code or existing systems
- The middleware or infrastructure that allows the discovery of services and the data exchange between application and services

[1] By end-to-end we mean the complete pathway through applications, the communication infrastructure and network, and the actual services to perform a specific task.

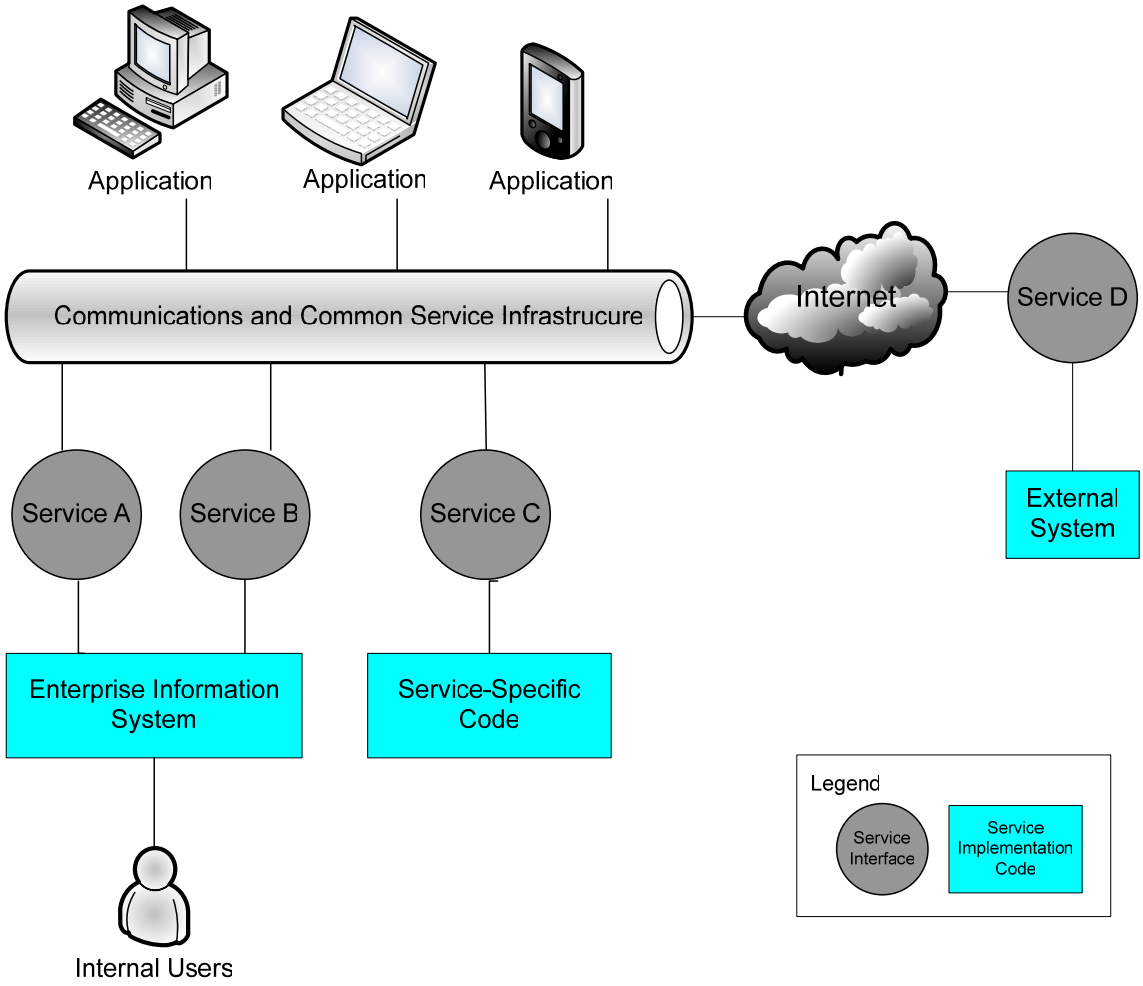


Figure 1. SOA-Based System

Developers of each type of component have different challenges:

Developers of applications that make use of services have to focus on the discovery and connection to services either statically at design time or dynamically at run time. The semantics of the information being exchanged is always a challenge, as well as what to do when a service is no longer available in the case of static binding. Their focus is on the discovery and connection to services - either statically or dynamically. Challenges include

- Identification of the right services
- Understanding the semantics of the information being exchanged
- Determination of rules to follow when services are no longer available (in the case of static binding)

Developers of services have to focus on the description and granularity of services so that applications can easily locate and use them with acceptable QoS. If services are going to be additional interfaces to existing systems, the focus is on the extraction of information from the systems and the wrapping of it within a defined message structure. If services are non-existent, the service-specific code needs to be written or reused from legacy systems. In this case, there are additional issues of service initialization. If reused, migration feasibility from legacy to target has to be analyzed. The challenges of service providers include

- Mapping of service requirements to component capabilities
- Description and granularity of services with acceptable QoS
- Determination of migration feasibility of potential services from legacy applications
- Writing new service specific code and wrappers

Developers of SOA infrastructure or middleware have to focus on providing a stable infrastructure as well as a set of common infrastructure services for discovery, communication, security, etc. Their focus is on providing a stable infrastructure. Challenges of infrastructure developers include:

- Providing common infrastructure services for discovery, communication, security, and QoS requirements.

Each group tends to focus on its own problems and build components to optimize its own concerns. This approach is detrimental to the engineering of end-to-end systems. How does an organization build an SOA-based system given these differences? How does it guarantee quality of service?

Security provides a good example of the problems raised by SOA in meeting quality of service requirements. One objective for an SOA is to easily tailor a computer-supported service for multiple business processes which reduces development costs and speeds deployment of systems for new or revised business processes. Although SOA identifies shared functionality, the multiple usages of a service may not have common security requirements. A service used for an internal business process has a very different threat model than when it is used to support an external business process. The implementation of an SOA has to support such variances in the non-functional requirements. The ad hoc management of the complexity generated by the connections among business processes, user actions, the system communication structures, and heterogeneous administrative systems certainly increases the probability of exploitable vulnerabilities.

Need for End to End Engineering

There are not good published strategies for building end-to-end systems based on SOAs. Most analyses focus on only one of the development perspectives and assumes an “ideal environment”. There is a need for a research agenda that examines each of the three perspectives: application developer, infrastructure developer, and service provider and answer question such as: How does one select the appropriate services and infrastructure for the organization’s system goals? How does one determine the quality of service delivered by a system when some of its components are discovered and composed at runtime? Addressing these issues can ultimately lead to an approach to building end-to-end systems based on SOAs.

Such as approach will focus on issues, such as such as

- Quality of service in a system where service discovery and composition plays a major role
- Insights on technology paths and limitations
- Information on quality of service needs
- Understanding of critical user needs in this area

References

[Ma 05] Ma, Kevin. Web Services: What's Real and What's Not. IT Professional. Volume 7 Number 2. IEEE Computer Society. March-April 2005.

[Manes 05] Manes, Anne. VantagePoint 2005-2006 SOA Reality Check. Burton Group. 2005