Special Issue: SAICSIT '99
Contents

Preface

P. Machanick ............................................................... 1

Research Articles

Active Learning: Issues and Challenges for Information Systems and Technology
RD Quilling, GJ Erwin and O Petkova .......................................................... 5

A Generic Modelling Framework for Interactive Authoring Support Environments
Paula Kotze ......................................................................................... 15

O Petkova and JD Roode ........................................................................ 26

An Information-Theoretic Semantics for Belief Change
T Meyer ................................................................................................. 33

A Complexity Metrics Model for Software Correction
A Törn, T Andersson and K Enholm .......................................................... 40

A Conceptual Design for High-Volume Data Processing of Warehouse Database into Multidimensional Database
Paisarn Trakulsuk and Vichit Avatchanakorn ............................................ 49

A Pragmatic Approach to Bitemporal Databases: Conceptualization, Representation and Visualisation
Chiyaba Njovu and WA Gray .................................................................... 58

A Building Recognition System
SP Levitt and B Dwolatzky ..................................................................... 68

Computer Programming and Learning to Write
John Barrow ............................................................................................. 77

Co-operating to Learn using JAD Technologies
TA Thomas ................................................................................................ 87

Critical Success Factors for the Implementation of DSS at a Selection of Organisations in Kwazulu/Natal
URF Averweg and GJ Erwin ................................................................. 95

Enhancing the Predictability of Two Popular Software Reliability Growth Models
Peter A Keiller and Thomas A Muzzuchi .................................................. 105
<table>
<thead>
<tr>
<th>Title</th>
<th>Author(s)</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalised Unification of Finite Temporal Logic Formulas</td>
<td>Scott Hazelhurst</td>
<td>110</td>
</tr>
<tr>
<td>Harmonizing Global Internet Tax: A Collaborative Extranet Model</td>
<td>E Lawrence and B Garner</td>
<td>119</td>
</tr>
<tr>
<td>Improving Object Oriented Analysis by Explicit Change Analysis</td>
<td>Lui Yu, Siew Chee Kong, Yi Xun and Miao Yuan</td>
<td>128</td>
</tr>
<tr>
<td>Reconciling the Needs of New Information Systems Graduates and Their Employers in Small, Developed Countries</td>
<td>Rodney Turner and Glenn Lowry</td>
<td>136</td>
</tr>
<tr>
<td>Shortest Delay Scheduling Algorithm for Lossless Quality Transmission of Stored VBR Video under Limited Bandwidth</td>
<td>Fei Li, Yan Liu, Jack Yiu-Bun Lee and Ishfaq Ahmad</td>
<td>146</td>
</tr>
<tr>
<td>Software Croma Keying in an Immersive Virtual Environment</td>
<td>Frans van der Berg and Vali Lalioti</td>
<td>155</td>
</tr>
<tr>
<td>Some Automata-Theoretic Properties of $\cap$-NFA</td>
<td>Lynette van Zijl and Andries PJ van der Walt</td>
<td>163</td>
</tr>
<tr>
<td>The CILT Multi-Agent Learning System</td>
<td>Herna L Viktor</td>
<td>176</td>
</tr>
<tr>
<td>The Development of a Generic Framework for the Implementation of Cheap, Component-Based Virtual Video-Conferencing System</td>
<td>Soteri Panagou and Shaun Bangay</td>
<td>185</td>
</tr>
<tr>
<td>The Role of Experience in User Perceptions of Information Technology: An Empirical Examination</td>
<td>Meliha Handzic and Graham Low</td>
<td>194</td>
</tr>
<tr>
<td>What are Web Sites Used for: Cost Savings, Revenue Generating or Value Creating?</td>
<td>Man-Ying Lee</td>
<td>201</td>
</tr>
<tr>
<td>New Ideas Papers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approaches to Video Transmission over GSM Networks</td>
<td>Bing Du and Anthony Maeder</td>
<td>210</td>
</tr>
<tr>
<td>From Information Security Baselines to Information Security Profiles</td>
<td>Rossouw von Solms and Helen van der Haar</td>
<td>215</td>
</tr>
<tr>
<td>Grounded Theory Methodology in IS Research: Glaser versus Strauss</td>
<td>J Smit</td>
<td>219</td>
</tr>
<tr>
<td>Introducing a Continuum of Abstraction-Led Hierarchical Search Techniques</td>
<td>Robert Zimmer and Robert Holte</td>
<td>223</td>
</tr>
</tbody>
</table>
Multimedia as a Positive Force to Leverage Web Marketing, with Particular Reference to the Commercial Sector  
**Stan Shear**  
229

Understanding HCI Methodologies  
**Peter Warren**  
234

Electronically Published Papers  

<table>
<thead>
<tr>
<th>Experience Papers</th>
</tr>
</thead>
</table>
| A Java Client/Server System for Accessing Arbitrary CANopen Fieldbus Devices via the Internet  
  **Dieter Bühler, Gerd Nusser, Gerhard Gruhler and Wolfgang Küchlin**  
  239 |
| An Object-Oriented Framework for Rapid Client-side Integration of Information Management Systems  
  **Ralf-Dieter Schimkat, Wolfgang Küchlin and Rainer Krautter**  
  244 |
| Distributed Operating Systems: A Study in Applicability  
  **Jürgen Prange and Judith Bishop**  
  249 |
| Formal Verification with Natural Language Specifications: Guidelines, Experiments and Lessons so far  
  **Alexander Holt**  
  253 |
| Introducing Research Methods to Computer Science Honours Students  
  **Vashti Galpin, Scott Hazelhurst, Conrad Mueller and Ian Sanders**  
  258 |
| Visualising Eventuality Structure  
  **ST Rock**  
  264 |

Electronically Published Papers  
SAICSIT’99

South African Institute of Computer Scientists and Information Technologists

Annual Research Conference 17-19 November 1999

Prepare for the New Millennium

Is there life after y2k?

Mount Amanzi Lodge, Hartebeespoort, South Africa

Sponsors

http://www.cs.wits.ac.za/

PHDS research/PHDS.html

Think different.
Preface

Philip Machanick, Overall Chair: SAICSIT'99

Running SAICSIT'99, the annual research conference of the South African Institute for Computer Scientists and Information Technologists, has been quite an experience.

SAICSIT represents Computer Science and Information Systems academics and professionals, mainly those with an interest in research. When I took over as SAICSIT president at the end of 1998, the conference had not previously been run as an international event. I decided that South African academics had enough international contacts to put together an international programme committee, and a South African conference would be of interest to the rest of the world.

I felt that we could make this transition at relatively low cost, given that we could advertise via mailing lists, and encourage electronic submission of papers (to reduce costs of redistributing papers for review).

The first prediction turned out to be correct, and we were able to put together a strong programme committee.

As a result, we had an unprecedented flood of papers: 100 submitted from 21 countries. As papers started to come in, it became apparent that we needed more reviewers. It was then that the value of the combination of old-fashioned networking (people who know people) and new-fashioned networking (the Internet) became apparent. While the Internet made it possible to convert SAICSIT into an international event at relatively low cost, the unexpected number of papers made it essential to find many additional reviewers on short notice. Without the speed of e-mail to track people down and to distribute papers for review, the review process would have taken weeks longer, and it would have been much more difficult to track down as many new reviewers in so little time.

Even so, the number of referees who were willing to help on short notice was a pleasant surprise.

The accepted papers cover an interesting range of subjects, from management-interest Information Systems, to theoretical Computer Science, with subjects including database, Java, temporal logic and implications of e-commerce for tax.

In addition, we were very fortunate in being able invite the president of the ACM, Barbara Simons as a keynote speaker. Consequently, the programme for SAICSIT'99 should be very interesting to a wide range of participants.

We were only able to find place in the proceedings for 36 papers out of the 100 submitted, of which only 24 are full research papers. While this number of papers is in line with our expectation of how many papers would be accepted in each category, we did not have a hard cut-off on the number of papers, but accepted all papers which were good enough, based on the reviews. Final selection was made by myself as Programme Chair, and Derrick Kourie, as editor of the South African Computer Journal. Additional papers are published via the conference web site.

We believe that we have put together a quality programme, and hope you will agree.

Acknowledgments

I would like to thank the South African Computer Journal production team, Andries Engelbrecht and Herna Viktor, respectively from the Department of Computer Science and Informatics, University of Pretoria, for their work on producing the proceedings.

The reviewers listed overleaf did an excellent job: many wrote very detailed reports, sometimes after being called in on very short notice. Inevitably, there were some glitches resulting from the unexpected workload, but the buck stops with the programme chair: I promise to do better next time.

I would also like to thank my own department for putting up with the extra work and expense that running a conference entails. I tried not to burden them with too much extra work, but our secretaries, Zain Gowar and Leanne Reddy, inevitably had to take on some extra work. John Ostrowick provided valuable assistance with design of our web pages and call for papers poster. Carol Kernick, who handles our finances and membership records, did a fine job of keeping up with the demands of the conference.

Finally, I would like to thank our sponsors, whose contribution made this conference been possible:

- PricewaterhouseCoopers - sponsored generous prizes and the conference banquet
- National Research Foundation (NRF) - provided financial support
- University of the Witwatersrand - provided financial support
- Programme for Highly Dependable Systems, University of the Witwatersrand - provided financial support
- Standard Bank - provided financial support
Editorial

• Apple Computer - provided equipment for the conference
• Qualica - provided technical support including helping with the conference web site

Web Site

For more information about SAICSIT, including a pointer to the conference site, see <http://www.saicsit.org.za>.

Referees

• Department of Computer Science, University of Pretoria
  - Derrick Kourie
  - Bruce Watson
  - Vali Lalioti
  - Andries Engelbrecht
  - Ivan Mphahlele
• German National Research Center for Information Technology - GMD
  - Gernot Goebbels
• School of Math Stat and IT, Natal University Pietermaritzburg
  - Peter Warren
• Graduate School of Business, University of Cape Town
  - Kurt April
• CSIR
  - James Jardine
• Department of Computer Science, Rhodes University
  - Shaun Bangay
  - Peter Clayton
  - John Ebden
  - Richard Foss
  - George Wells
  - Peter Wentworth
• Information Technology Division, Rhodes University
  - Caro Watkins
• Department of Information Systems, Rhodes University
  - Brenda Mallinson
  - Dave Sewry
• Department of Informatics, University of Pretoria
  - Carina de Villiers
  - Herna Viktor
  - Niek du Plooy
  - Elsje van Rooyen
  - Machdel Matthee
  - Alan Abrahams
  - Jackie Phahlamohlaka
• School of Information Technology & Engineering, University of Ottawa
  - Stan Szpakowicz
  - Dwight Makaroff
• Department of Information Systems, Victoria University
  - Glenn Lowry
  - Peter Shackleton
  - Tas Adam
  - Alastair Wallace
  - Stephen Burgess
  - Dave Burgess
  - Julie Fisher
  - Jerzy Lepa
  - Geoff Sandy
  - Rod Turner
  - Alastair Wallace
  - Andrew Wenn
• Peninsula School of Computing and Information Technology, Monash University
  - Ainslie Ellis
• School of Information Management and Systems, Monash University
  - Angela Carbone
• School of Computer Science & Software Engineering, Monash University
  - Dianne Hagan
  - Judy Sheard
• School of Management Information Systems, Deakin University
  - John Lamp
  - David Mackay
  - Philip Joyce

SACJ/SART, No 24, 1999
Editorial

- Marie van der Klooster
- Bill Hewett
- Jennie Carroll
- Rodney Carr

• Computer and Automation Research Institute, Hungarian Academy of Sciences
  - Ferenc Vajda

• Department of Computer Science, University of the Witwatersrand
  - Ian Sanders
  - David Lubinsky
  - Conrad Mueller
  - Yinong Chen
  - Bob Baber
  - Vashti Galpin
  - András Salamon
  - Scott Hazelhurst
  - Philip Machanick
  - Zoltán Fazekas

• Department of Electrical Engineering, University of the Witwatersrand
  - Barry Dwolatzky
  - Farzin Aghdasi

• Department of Mathematics, University of the Witwatersrand
  - Jonathan Burgess

• School of Law, University of the Witwatersrand
  - Victoria Bronstein

• Edward Nathan & Friedland Attorneys, Johannesburg
  - Justine White

• Department of Information Systems & Technology, University of Durban-Westville
  - Geoff Erwin

• Department of Computer Science, Clemson University, Clemson, SC
  - Karen Hay

• Logical SA
  - Philip Green

• Department of Computer Science, University of Bristol
  - Alan Chalmers

• Department of Systems and Computer Science, Howard University, Washington, D.C
  - Todd Shurn
  - John Trimble

• Department of Computer Science, University of Cape Town
  - Sonia Berman

• King’s College, London
  - Peter Wood

• Department of Information Systems, University of Cape Town
  - Paul Licker
  - Steve Erlank
  - Alemayehu Molla

• Department of Department of Mathematics and Applied Mathematics, University of Cape Town
  - Ingrid Rewitzky
  - Peter Jipsen
  - Renkuan Guo

• Department of Information Systems, University of the Western Cape
  - Andy Bytheway

• Department of Electrical Engineering and Computer Science, The George Washington University, Washington, DC
  - Rachelle Heller

• School of Biophysical Sciences and Electrical Engineering, Swinburne University of Technology
  - Ian Macdonald

• Electrical and Electronic Systems Engineering, Queensland University of Technology
  - Anthony Maeder

• Computer Science Department, Abo Akademi
  - Aimo Törn
  - Kaisa Sere
  - Mats Aspnäs

• Division of Informatics, University of Edinburgh
  - Graeme Ritchie
  - Arturo Espinosa-Romero
  - Jon Oberlander

SART/SACJ, No 24, 1999
Editorial

- Chris Brew
- Alexander Holt

• Department of Computer Science and Engineering, Helsinki University of Technology
  - Tommi Junttila
  - Nisse Husberg

• Technical University of Budapest, Department of Measurement and Information Systems
  - Andras Pataricza

• NASA-Goddard Space Flight Center
  - Nigel Ziyad

• Department of Computer Science & Information Systems, University of Port Elizabeth
  - Janet Wesson
  - Leon Nicholls

• Computer Technology Department, Indiana University Purdue University Indianapolis
  - Tim Price

• Department of Electrical Engineering and Computer Science, The University of Michigan
  - Trevor Mudge

• Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
  - David Forsyth

• School of Computer Science, University of Birmingham
  - Mark Ryan

• Faculty of Mathematics, Computer Science, Physics & Astronomy, University of Amsterdam
  - Carlos Areces

• Department of Computer Science, Universidade do Vale do Rio dos Sinos - UNISINOS Rio Grande do Sul
  - Marcelo Walter

• School of Information Technology & Mathematical Sciences, University of Ballarat
  - Binh Pham
Improving Object Oriented Analysis by Explicit Change Analysis

Liu Yu, Siew Chee Khong, Yi Xun and Miao Yuan

School of EEE, Nanyang Technological University Nanyang Avenue, Singapore 639798, eyliu@ntu.edu.sg

Abstract

Changeability is one of the major concerns in software development. OO (Object-Oriented) technology itself is not enough to solve this problem, i.e. software systems can not have enough changeability by just using any specific kind of OO language or computing models such as CORBA or COM. In this paper, we first discuss how the changeability issue is handled in current software development methods and what is the most important prerequisite for solving the software changeability problem. Based on these analyses, we argue that most of current OO analysis methods have missed an important activity in their analysis phase, i.e. change analysis. Thus, OO software systems developed according to these methods often can not exhibit the expected maintainability, extensibility and reusability. In order to improve current OO analysis methods, we have proposed a conceptual framework for conducting a systematic change analysis in software analysis phase. A detailed case study of the application of this framework in a real software project has also been described in this paper.

Keywords: Object-orientation, Change Analysis, Software Engineering, Changeability

Computing Review Categories: D.2.1

1 Software Changeability

Software products are quite different from hardware ones. Changeability may be the most important difference between them. Software has kept on changing during its entire maintenance phase. Hardware maintenance is much different from software maintenance. Hardware maintenance is the process of retaining a hardware system or component in, or restoring it to, a state in which it can perform its required function [1]. But software maintenance is a process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [1]. Software has to experience more change than hardware during its life cycle because sometimes software is the only changeable part after the whole system has been delivered. Software is always subject to the pressure of change. When there is any new requirement or proposed change from users to the current system, the software part in it will be changed at most of time rather than its hardware part. In fact, software is often used to provide more changeability to a system. At many systems, when more changeability such as adaptability, flexibility, and extensibility is needed, a software-based implementation is often used to replace its original rigid hardware-based implementation. In addition to its maintenance, software also needs much changeability to support software reuse. Several researchers [5, 6] have identified that “as-is” reuse is very unlikely to occur and that in the majority of the cases, a reused component has to be adapted or changed in some way to match different application’s specific requirement. Any reusable software component should be able to be changed quickly and cheaply along some predetermined directions. If the cost of making this change is greater than that of writing a new component, nobody will reuse it. In practice, the cost of software changes made in maintenance phase is very high compared with its initial development cost. According to Stephen R. Schach [2], at least about two-thirds of total software costs are devoted to maintenance changes. Therefore, more money has been spent on changing software in its maintenance phase than on building it in its development phase. From above discussion, we can see that changeability is a very important issue to software. Software change is costly but inevitable. Many non-functional requirements on software, such as maintainability, adaptability, flexibility, extensibility and reusability, can be seen as different aspects of software changeability respectively.

2 Design For Change

How do our software engineers cope with the changeability problem in software? “Design for change” [7, 8] may be the only effective solution to it. Actually, "design for change" had been proposed and well explained for a long time [15, 23, 24], but still has not been well-practiced [24]. Also more and more people have agreed that the maintainability of a software system is dependent on its design [35]. Generally, there are two different kinds of possible changes in software, i.e. anticipated and unanticipated changes. Software can not be changed easily unless these changes have been anticipated beforehand. One of the clearest morals about “design for change” as it is taught in other areas of engineering is that one must anticipate changes before one begins the design [15]. The “Design for change” principle is applicable only to anticipated changes. “Design for change” is trying to encapsulate the...
areas of a system that are likely to change and provide non-intrusive ways for a later designer/programmer to modify the behavior of the code [7]. Information hiding [26] is the core of the "design for change" principle. The first crucial step in information hiding is to identify the items that are likely to change. These items are termed "secrets." And then, designing inter-module interfaces that are insensitive to the anticipated changes [15]. Instead of ready to be changed arbitrarily, software can be changed easily only in some specific manners or through some specific mechanisms that are pre-defined by or embedded in some specific places in software by its designers. "Design for change" just means "Design for anticipated change" instead of "Design for all change" because only anticipated change can be effectively dealt by it. OO technology only can be used to cope with anticipated changes as well. It will provide software with some specific kinds of designated change-ability, but not arbitrary flexibility and adaptability. Using OO can not guarantee that the final system will be generally flexible, adaptable, or reusable. In fact, changeability must be explicitly engineered into software. Software systems are adaptable in specific, designated ways, if at all, to some specific kinds of change [12]. For example, OO Design Patterns [8] only can be used to help software engineers avoid system redesign by ensuring that a system can change in some specific ways, not arbitrary ways. Each design pattern permits some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change. Different patterns will support different kind of anticipated change. To design a system to be robust to potential changes, you must first predict what kind of change might happen over its lifetime [8].

3 Why We Need Change Analysis?

Software changeability is always thought to be an issue which should be handled in software design phase. From many discussions of "design for change" [15, 23, 24, 7, 8], we can find that the problems related to software change-ability are completely left to software designers to cope with. Since only anticipated changes can be effectively handled in software design phase, so the most important requisite for applying "design for change" philosophy is that software designers must know what kind of future change should be supported by the system before he can start his work on design. Only after he knows the potential future changes in system, a software designer is able to design or embed some corresponding change-dealing mechanisms into software to properly encapsulate these possible changing areas and provide some specific non-intrusive way to ease maintainer to do proper maintenance changes in future. In fact, the success of "design for change" heavily depends on our ability to predict the future [24]. Effective modularization relies on the predictability of future changes [27]. But the question is how software designers get to know these potential changes in software design phase? Designers can not effectively predict and identify the future potential changes in system by themselves in the design phase. Many kinds of software changes are caused by changes in its original software requirements. Most software requirements changes in future are often closely related to various factors in its application domain. But software designers normally do not have enough specialized application domain knowledge to help them make a correct and precise prediction of future potential changes in current system requirements. In almost all software development methods, the only output from software analysis phase, which is also the input to its design phase, is the software requirements specification. So the only channel for software designers to get the knowledge about the anticipated changes in user's requirements on system is through the requirements specification documentation. In order to design a system based on the principle of "design for change", the description of anticipated future changes in requirements should be one part of the software requirements specification. And correspondingly, some kind of activity in the analysis phase should be carried out to collect the information about the potential changes in current requirements to system. This particular activity in the analysis phase, which is performed to determine the potential changes in the current user requirements on system, is called change analysis by us. Unfortunately, almost all current available OO software development methods have ignored this prerequisite for applying the "design for change" principle in software design phase, and the change analysis has not been included as one part of their analysis activities. In fact, the consideration of potential changes in system should be started as early as from the software analysis phase, and it can not afford to be delayed until to the design phase, which is exactly what happens in most current OO software development methods. Maintainability, adaptability and extensibility should not be only the concerns of software designers and programmers. Software analysts must also assume his part of responsibility, i.e. conducting a proper change analysis in software analysis phase to provide designers the necessary information of the future potential changes in current requirements. This information will help designers to construct a maintainable and extensible system, i.e. a system that has proper pre-defined change-dealing design structures and mechanisms to accommodate future potential changes. Change analysis should become an inseparable step in the analysis phase of any software development method which aims to produce a high-quality software product with good maintainability and extensibility. A successful change analysis is the very first step to get a highly maintainable system architecture which is able to accommodate future maintenance changes quickly and cheaply.
4 Problems In Current Object-Oriented Analysis Methods

Many object-oriented software development methods have been proposed [22, 19, 28, 29, 30]. Smooth transition from the analysis model to its design model is often claimed to be a prominent characteristic of most OO methods. They suggested that the design object model could be directly derived from their analysis object model simply by adding some implementation details, and not many efforts need to be put into their design phase. That is why most OO methods do not have much discussion on their design phase and only concentrate on the analysis phase. So this paper will also mainly discuss the problems in the analysis phase of these OO methods. The common problem of them is that most of these OO methods have not given an appropriate consideration to software changeability issue in their analysis phase. These methods only concentrate on how to satisfy the current user requirements, but not pay enough attention to the possible future changes in system, so the “design for change” principle can not be well supported by them. A direct result of this negligence is that the system design structure constructed according to these OO development methods usually do not have the expected extensibility, flexibility, adaptability, maintainability and reusability, which are often thought to be the most important advantages of OO technology and development methods. These OO methods advocate that their analysis object model mainly consists of the objects and relations that are directly mapped from its original application domain. But almost no consideration has been given to the possible future changes of these application objects and relations. So this kind of analysis model is at best a precise model for user’s present requirements, but can not easily be adjusted to tolerate user’s future changing requirements. The corresponding design model that is “smoothly” transformed from the analysis model can not have much changeability to cope with the potential changes in system either. This analysis model derived from these OO analysis methods is barely able to cover the representation of current requirements on system, but never can get a chance to be a basis for a successful design object model which is flexible, adaptable, and robust enough to deal with most potential future changes easily. This kind of analysis model is more close to the problem domain rather than the solution domain. Since the potential changes in requirements has not been reflected in the analysis model, the design model directly transformed from it will not contain any appropriate change-dealing structure and mechanism to support future changes in system. The software developed according to these OO methods will not have the expected extensibility, reusability and maintainability. Let us check several mainstream object-oriented analysis methods: In OOA [28], the analysis activity has been divided into several phases, i.e. finding class-&-objects, identifying structures, identifying subjects, defining attributes, defining services etc. Although “continual change” has been listed as one of analysis challenges, there is no specific instruction or guideline on how to deal with this challenge in the analysis phase. The only available suggestion was that the analyst should seek the requirements packaging that will remain stable over time. No other detail has been given. In Booch method [19], the purpose of software analysis phase was defined as to provide a model of future system’s behavior. Milestones of the analysis phase are scenarios for all fundamental system behaviors. Two primary activities are associated with the Booch’s analysis method: domain analysis and scenario planning. The domain analysis concentrates on studying existing systems. The scenario planning will identify all major system operation scenarios. But no specific action or step has been defined to identify or analyze the potential changes of these scenarios. In Fusion [22], the goal of its analysis phase is to capture the user requirements as much as possible and then represent it in a complete, consistent, and unambiguous format. According to the Fusion, most input to the software analysis phase usually is a natural language requirements definition document. Such kind of document often contains the description of what the problem is, what some solutions might be, and how a system that solves the problem ought to work. They often contain ambiguity and are at best only partially complete. At worst they can be internally inconsistent. The analysis phase in Fusion only concentrate on how to make vague user requirements more clearly and consistently. No discussion relevant to potential requirement changes has been mentioned in Fusion. There is a quite similar situation in OMT [29] as in Fusion. The main purpose of OMT’s analysis phase is to make the general problem statement from clients more precise, and expose ambiguities and inconsistencies in it. Its analysis model is intended to be a precise and concise representation of the problem that permits answering questions and building a solution. Then the subsequent design steps can refer to the analysis model, rather than to the original vague problem statement. So the main focus of the OMT’s analysis phase is still on re-formulating a natural language requirements statement with its object modeling structures. Once again, the future possible changes in current requirements have been ignored almost completely. In conclusion, most of available object-oriented analysis methods only discuss about how to collect and represent the current user requirements with their different object models, but do not care about the future potential changes in these requirements, and none of them suggests to do a systematic change analysis in their analysis phase.

5 A Framework For Conducting Change Analysis

The change analysis will begin only after the current user requirements to system have been gathered. It should be a systematic process to identify and categorize all those meaningful and probable potential changes in user’s requirements to system. These potential changes in the user’s requirements will be most likely to directly cause the future
maintenance changes in the final system. The proposed framework for this change analysis consists of following steps:

1. Make clear the life span for the system being analyzed. The results of change analysis will be effective only for this period of time. Since the change analysis actually will predict the future potential changes in the user requirements, uncertainty is an important factor to be considered. The changes that should be taken into account in the change analysis are those possible maintenance changes that will happen during the expected lifetime of this software. Beyond its intended life span, there is no point to waste any resource to predict anything. Also beyond certain time horizon, anything in system will not be able to be effectively predicted because of too much uncertainty. So the first thing we should do in the change analysis is to determine the prediction time horizon for it.

2. Identify and classify potential future changes. The changes in requirements are often resulted from some changes in its original application domain. There are many external factors that can cause software to change, but most of them can be classified into several common categories, for example, changes in technology, changes in its running contexts, social and organizational environments, changes in business law and company policy etc. In fact, we can use these common categories as a hint for us to identify the future potential changes in any new system and classify the identified changes according to them. We have determined the following common categories of factors which may cause the potential changes in user's requirements to system:

- External interface change.
  - Interface change to external hardware. This kind of change can be the result of changes in the running machine, peripheral hardware such as display, printer and networking devices etc. The change may involve in the number, type and parameter of hardware interface
  - Interface change to other software.
  - User interface change. This kind of change can be found almost in every application.
  - Change in input and output data, such as its format, length, precision, unit, quantity etc.
- Function or sub-step variability
  A system function or its sub-function, and one step or its sub-step in a system's application logic may be implemented in a different manner in future, though it may still perform the same basic function or processing. For example, the current communication between two distributed parts of a system is implemented by a RS232 serial port connection. But this implementation is very likely to change, e.g. the two parts may also be linked by a dial-up line, LAN or WAN in future when the system works in a different environment.
- Change in processing strategy and control algorithm.
  For instance, the current encryption algorithm, resource scheduling and error-handling strategy specified in the requirements document may be replaced or adjusted in future due to the increased security concern, changes in available resources, and the new request on a system's reliability.
- Adding new function or service.
  Normally, new business service or new application function has to be introduced into system continuously in order to satisfy the user's increasing demand on current system. Software can not support for adding any new function freely into system. Only the new function, which has been anticipated beforehand in the analysis and design phases or is at least compatible with anticipated changing directions, can be combined into the current system seamlessly.
- Change in the user's requirements on the throughput, performance, efficiency, speed, accuracy etc. of the system.
- Rule and policy change.
  For example, business rules such as when to reorder inventory or the meaning of an aged account is often subject to change over a business application's life cycle. Changes in legal requirements arising from domestic legislation, from international regulations and standards will also result in this kind of change.
- Change in the executing sequence or invocation order in application logic.
  The executing sequence of sub-steps or sub-functions in a system's application logic or in a business process is often subject to change. This kind of change often will result in the change in the system's control flow or its function and sub-function invocation order.

3. To each kind of change identified in step 2, we should first determine how much impact it has on the proposed system. Some kind of potential changes can be more influential. For example, in a business information system, the transaction rule or policy change caused by the adjustment in legal requirements may be mandatory and have to be implemented according to a strict externally determined timetable. If the change is not supported in time, the organization concerned may be subjected to legal sanctions.

4. Determine the probability and frequency of one particular change. If one kind of change has great impact on
software system, but only has very small probability, we may still decide not to support it in our system. The future change with potential high frequency should be given a high priority. The ease of making a change in the design should bear a reasonable relationship to the likelihood and frequency of the change being needed [10].

5. According to the impact and probability of each potential change, we finally determine its priority to be given in the future system design. Only the probable change with enough priority will be supported by the system.

6 Case Study

This was a real information system development project completed by six Master students. The system is now running in a large electronics company, CET Technologies Pte Ltd, in Singapore, to automatically manage its electronic devices located throughout Singapore. Initially, the system was proposed by and developed for a small division of CET and intended to manage the devices only running in this small division. Other divisions had expressed some interests to the system. But during its software analysis phase, the students were asked by us to do a systematic change analysis to identify possible and meaningful changes in the requirements to system. Then after one year since the system was deployed, CET decided to use the system to manage all its similar devices in company. Although this decision had resulted in many changes to the early version of the system, it still had been successfully maintained to support almost all newly proposed revisions.

Major changes in the original requirements had been implemented relative quickly and cheaply because most of them had already been identified in the change analysis and given some proper consideration at design time. The original system design model had showed good extensibility and adaptability during this maintenance effort. This project is a good example to illustrate how the change analysis results can be used to improve the OO analysis and design object model, which has been developed with traditional OO methods, to make it more maintainable and extensible. In order to explain our ideas succinctly, we have simplified the original system and only selected a small part of it to discuss. The earliest version of the user requirements to the system which was intended to run in the small CET division can be summarized as follows: An information system is needed by Division M in CET to automatically monitor and control 18 devices running in its device room. The devices are the same kind of electronic hardware equipment and can be directly connected to the serial port of a computer through a multiplexer. The first system error is detected, the device manager will record it in a log file. Then the manager will generate a kind of alarming signal by the user interface, e.g. alarming sound or a pop-up dialog box to display the error information such as its equipment ID, error code, time stamp etc. Depending on its error code, the device manager will try to correct some kinds of errors by changing corresponding settings of the erroneous device. The new settings can be transmitted to erroneous device through a multiplexer. The first system design object model, which was transformed directly from its analysis object model and no change analysis had been conducted, was quite straightforward and simple. This object model was mainly made up of application domain objects. The simplified class diagram of this design is shown in Figure 1 with OMT notations. There were only two classes, i.e. CETEqp and Manager. The CETEqp class representing running devices in CET that need to be managed. Manager object would automatically poll all CETEqp objects for their running status. When any error occurs and is detected, Manager will try to recover it by transmitting new settings to it. Here a Manager object will manage all CETEqp objects.

If the final system had been implemented according to the design model in Figure 1, it would have had little maintainability to cope with the following maintenance changes. Any small change in user's requirements might cause rippling effects in many other parts of system. Only the user's requirements at that particular moment had been considered in this model. Almost no attention had been paid to its potential changes. Then we asked the students to conduct a systematic change analysis according to the aforementioned analysis framework. The students had discussed carefully with CET domain experts for several times to collect information about potential changes in their original requirements specification. Students had modified the old object model in Figure 1 to reflect their change analysis results. The revised object model is shown in Figure 2.

We will give some explanation of the object model in Figure 2 according to relevant change analysis results. It was just these potential changes identified from the change analysis that had resulted in all differences between Figure 2 and Figure 1. The following is some potential changes identified by students and their corresponding effects on the original object model.

1. The type of managed devices was very likely to change in future. The division in CET might also use other new or some different kinds of electric devices in future, and each of them had to be managed by cur-
current device manager in system. In order to support this change in device type, an abstract device class CETEqp had been introduced and served as the base class for all other concrete device classes such as Eqp1 and Eqp2, which represented different kinds of CET devices. The CETEqp class actually had defined an abstract interface and a standard interaction protocol between Manager object and all different kinds of device objects. The variations in device types were hidden behind the interface.

2. The location of CET devices was also likely to change when more devices were deployed by CET in future. Some devices may have to run remotely instead of working locally some time later. In fact, it was not necessary for Manager class object to know whether a managed CET device was a local device or a remote one as long as the device could support the predefined CETEqp interface and their interaction protocol. In order to keep this potential change of location from affecting other parts of the system, two new classes had been defined, i.e. LocalEqp and EqpProxy. The LocalEqp class was the base class for all devices running locally. The EqpProxy class was actually a local representative of a remote device and was used to hide the location difference between different devices.

3. Different remote CET devices running in different places might use different mechanisms to communicate to its device manager in future. In order to support this possible change in communication modes, a new class Communication, was defined to set up a common interface for different communication ways. Each subclass will support a specific kind of communication mode such as dial-up or networking.

4. The device polling strategy might also be changed in future. Currently, a Manager object would poll all its managed CETEqp device objects in a round robin manner for their status information. But as more and more different types of devices were to be managed by this Manager at the same time, the round robin polling strategy might not scale well. For example, some kind of mission-critical task might be running on some CET devices. The delay in detecting and correcting its errors could be disastrous. These devices should be treated differently from the others, e.g. should be checked more often than other ordinary devices. So under this circumstance, a priority-based polling strategy was preferable to a simple round robin polling strategy. In fact, what kind of polling strategy would be adopted was mainly decided by what types and how many CET devices should be managed in current system. There was no polling strategy that could work well in all situations. Sometimes different polling strategies have to be interchanged dynamically. So the potential change in polling strategy is almost inevitable and must be well supported. Here the Strategy design pattern was adopted to provide needed flexibility. The Manager, PollingStrategy, RoundRobin and Priority classes corresponded to Context, Strategy and ConcreteStrategy classes in Strategy pattern respectively.

5. The number of managed devices might increase in future. It could be thought as a change in system's throughput. Since all CET devices were managed by only one Manager in Figure 1, the manager was very likely to become a bottleneck of the whole system in future. In order to cope with the potential increase in system's throughput in future, a hierarchical system structure was used here to make system more scalable. As shown in Figure 2, there could be several Manager objects in this object model instead of only one in Figure 1. These Manager objects had been organized as a hierarchical structure and each of them was at a specific hierarchical level. Based on this new structure, a Manager object could manage many CETEqp device objects as well as several other lower-level subordinate Manager objects, but it only had at most one upper-level Manager object. Any subordinate Manager object could be treated the same as a CETEqp device object by its upper-level Manager object. Thus, the overall workload of device management had been distributed among different Manager objects located in different levels. Then a new class CETNode was defined and became the superclass of CETEqp and Manager classes. The common interface defined in CETNode will support all basic polling, setting and trapping functions.

Here we have discussed only five different kinds of identified potential changes and their effects on this sys-
tem's design structure. The others have been omitted due to limited space. Based on all identified potential changes in system, a design model in Figure 2 had been finally derived. It was more complex but also much more maintainable and extensible than its old counterpart in Figure 1. At most conditions only one or two classes in Figure 2 were affected when changes occurred during its maintenance phase in our project. Note that the object models in Figures 1 and 2 have been tailored from the real object models in project. Many other issues such as distribution, currency, and exception handling are not discussed here.

7 Related Works

Change case [34] can be used as a specific format to represent our change analysis result. Their work has concentrated on how to describe the identified future change with change case and how to use it to measure the impact of a change to other parts of system. But this article did not explain why and how to identify a potential change from the analysis phase in a systematic way. Most of the current work on software changeability has concentrated on impact analysis [36, 37, 38, 39, 40, 41], i.e. identifying software workproducts affected by a potential software change and assessing the consequences of a change. But impact analysis itself does not study how to improve the changeability in a software system. In fact, most impact analysis is performed after the development work in a system has been completed, or at least its design has been finished. What we are concerned in the change analysis is how the software analyst should do in the analysis phase in order to support the software designer to make correct design decisions on changeability, i.e. provide the system with appropriate changeability to cope with future maintenance changes efficiently and quickly.

8 Conclusions

Growing maintenance costs have become a major concern for developers and users of software systems. Changeability is one of important factors that have a great impact on software maintenance costs. Although OO methods were thought to be able to provide enough changeability to software, many completed real OO systems developed according to these methods do not exhibit as much as the expected extensibility, adaptability and reusability etc. One of the important reasons for this problem is that most OO analysis methods only concentrate on how to collect and represent the user's present requirements, and fail to conduct a systematic change analysis to identify and analyze the potential future changes in current requirements to system. Although the "design for change" principle has been proposed for many years, it still has not been well accepted by and integrated with most mainstream OO development methods. In fact, just like "design for change" must be a essential part of any software design methodology, the change analysis should also be included in any software analysis methods, whether it is OO or not.

References

[10] PerOluf Bengtsson and Jan Bosch, Scenario-based Software Architecture Reengineering, http://www.ide.hk-r.se/[


Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research notes. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications of Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript
Manuscripts for review should be prepared according to the following guidelines:

- Use wide margins and 1½ or double spacing.
- The first page should include:
  - the title (as brief as possible)
  - the author's initials and surname
  - the author's affiliation and address
  - an abstract of less than 200 words
  - an appropriate keyword list
  - a list of relevant Computing Review Categories
  - Tables and figures should be numbered and titled.
- References should be listed at the end of the text in alphabetic order of the (first) author’s surname, and should be cited in the text according to the Harvard method.

Manuscripts accepted for publication should comply with guidelines as set out on the SACJ web page, http://www.cs.up.ac.za/sacj

Please see the web site for further information on how to produce manuscripts which have been accepted for publication.

Authors of accepted publications will be required to sign a copyright transfer form.

Charges
Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect typesetting, reproduction and other costs. Currently, the minimum rate is R30.00 per final page for contributions which require no further attention. The maximum is R120.00, prices inclusive of VAT.

These charges may be waived upon request of the author and the discretion of the editor.

Proofs
Proofs of accepted papers may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Letters and Communications
Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words. Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book Reviews
Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement
Placement of advertisements at R1000.00 per full page per issue and R500.00 per half page per issue will be considered. These charges exclude specialised production costs, which will be borne by the advertiser. Enquiries should be directed to the editor.