Conference Proceedings

January 19-21, 2006, Montreal, Canada

Editor
Ashraf Gaffar
Papers and Academic Presentations

CUSEC 2006

Canadian University Software Engineering Conference

Engineering
Useful
Software

January 19-21, 2006
Montreal, Canada

Platinum Sponsor

www.microsoft.com

As an industry leader in the design and development of innovative computer technology, Microsoft's efforts are reflected in a range of technological advances in areas such as operating systems, software development platforms and tools, Web services, knowledge management, natural language processing, privacy, security, and networking.

Imagine having the resources to influence tomorrow's reality today, and having fun while you do it. That's Microsoft. Right now, we're looking for people who think big and dream big – people a lot like you. If you're ready to discover just how far your talents can take you, we invite you to Microsoft. From there, how far you go is up to you.

Silver Sponsors

www.ea.com

www.gameloft.com

Friend Sponsors

www.pearsoned.ca/highered

www.oreilly.com/

Academic Sponsors

http://www.encs.concordia.ca/

http://www.mcmaster.ca/

Department of
Computer Science and Software Engineering
http://www.cs.concordia.ca/

Table of Contents

II Corporate Presentations

III Tutorials

IV Papers

1- Software Modeling and Methodologies

2- Software Methodologies

3- Design Tools

4- Multimedia

5- Society and Education

CUSEC Committee Members

Chair	John Jonathan Kopanas
Conference Advisor	Dr. Peter Grogono
Finance	Mark Pavlidis
Corporate Sponsorship	Neeraj Mathrani
Keynote, Tutorial and Corporate Presentations	Ahmed Kamel
Papers and Academic Presentations	Ashraf Gaffar
Promotion	Michel Parisien
Logistics, Website and Registration	Nadia Chaouch

Special Thanks to:

Audrey Girouard, École Polytechnique de Montréal

Ethel Macasias, McMaster University

Opening Remarks

The Canadian University Software Engineering Conference (CUSEC) is happy to celebrate its 5th anniversary. All this would not have been possible without your constant support. So, first we would like to say thank you. Thank you for your ever growing interest in the field of Software Engineering.

The broad view of the conference remains the same as before: to examine the future of software engineering.

The conference brings together researchers, academics, industry practitioners and students in order to share ideas, expand engineering knowledge, and introduce well-established techniques. The idea of the conference stemmed from the need of passionate software engineers to unite with others to promote software engineering; the desire to attract bright minds to research software engineering; and the demand for bringing people together to discuss how theory is implemented. Therefore, our priority is the promotion and development of software engineering as a new discipline. Offering students, as well as teachers and professionals, a conference that combines purely scientific considerations with the industrial practical applications in the effervescent field of software engineering is what distinguishes us from other conferences.

The conference is synonymous to new knowledge and personal enrichment. It is an occasion for passionate presenters to communicate their thoughts and works to an equally passionate audience. It also enables our corporate partners to make their companies known, to meet future employees, and to be associated to a product of quality, innovation, and reliability.

The chosen theme this year is “Engineering Useful Software”. The conference has traditionally included corporate, academic and paper presentations. This year we will also incorporate tutorials, labs, as well as Career Fair and Corporate Expos.

The organizing team would like to welcome you to CUSEC 2006, hoping we enjoy it all together until the very last minute.

Sincerely,
Ashraf Gaffar
On behalf of the CUSEC Team

Chair's Remarks

We are celebrating our 5th year anniversary this year, but it seems like only yesterday that I was chairing our first conference. A conference that almost never happened.

Five days before our inaugural event we only had four people registered. We could of packed up our bags and chalked this one up as a loss, but we did not want to prove our critics right. Many people said that students, especially undergraduate students, had no business organizing an academic conference. What thing everyone should remember is that the more nay-sayers you have, the more you should realize that you are headed in the right direction.

With only five days left, we worked night and day coming up with new marketing material, distributing it, making class presentations, getting deals done with organizations to subsidize tickets, whatever we could think of we did and at the end of the day we got over 100 people to sign-up in less then five days.

Every single person on the founding team learned an important life lesson. If you really want something bad enough… it's yours. Not only did the founding members of CUSEC learn a valuable lesson about life, they gave something back to the software engineering community in Canada. Something that is worth more then the lessons they learnt, it is unfortunate though because I don't think any of them realize it – they gave us CUSEC.

There was nothing inherently special about any of the founding members, myself included. They were your regular students, going to school, trying to better themselves. What brought these unsuspecting students together though was a passion, a passion for software engineering, a passion that they wanted to share with their peers.

If you are reading this, it is because you have something in common with all of CUSEC's founding members. And like those members, you can leave your mark. It takes many volunteers to put CUSEC together. Volunteers who give their scarce free time to CUSEC. But I can guarantee you, if you ask any of them, none of them would trade their CUSEC experience for the world.

The opportunity to bring your favorite software engineers from around the world together under one roof. To be apart of something that most people thought to be impossible for undergraduate students. To make new friends, friends that will always occupy a special part in your heart.

CUSEC has created a special experience for Software Engineering students. As the field of Software Engineering grows, your thought, ideas, and discussions here at CUSEC will help to shape its future. Not only through the interaction with speakers who are leaders in the field, but with each other – the future leaders in the field.

Every student leaves university with a degree. But not all of them can say they left their mark. If organizing CUSEC is not for you, the only thing I ask you to do is take a couple of minutes, hours or even days, reflect on how you want to be remembered, make a plan, and go for it. If you don't think you owe it to yourself, then you at least owe it to the people around you.
If you do decide to be apart of CUSEC all I want to do is leave you with one thought. Your reward is priceless, but if you don't pay attention at just the right time you might miss it. The first day of CUSEC stop and watch. Look at all those smiling faces… could you ask for anything more.

Your Friend and Fellow Software Engineer,
John Kopanas

(I want to take this opportunity to mention the founding members, they gave us much more then they could of ever realized at the time: Chae Dickie-Clark, Madelaine Tang, Marc Abbyad, Chadi Freeha, Jason, Jacinthe Gagnon, Dr. Peter Grogono and John Kopanas.)

Keynote Speakers

Modular Concurrency

Peter Grogono
Professor and Associate Chair, Concordia University

Abstract:
Software development moves onwards and upwards to ever-higher levels of abstraction, further and further away from the code that actually runs on the hardware. This is a good thing – the more details we can hide the better – but it conceals the unfortunate fact that modern programming languages match neither current software requirements nor the underlying architecture. Significant contributions to concurrency and encapsulation, made during the 1970s and 1980s, have been lost in the excitement of programming with objects and networks. Programming languages remain the basic tool of the software engineer and they must provide the reliability, security, and performance that modern applications require. In the talk, I will propose a new programming paradigm designed to meet the needs of the next generation of software. The key features of the paradigm are strong encapsulation, full concurrency, malleability, and scale-free design.

Biography:
Peter Grogono built his first computer when he was fifteen. After obtaining a mathematics degree from Cambridge, he accepted a post as a mathematician but quickly replaced the Monroe calculator with FORTRAN. After spending a few years dabbling in operating systems, engineering, electronic music, and accounting systems, he joined Concordia University as a systems analyst. In 1984, he moved from the Computer Center to the Computer Science Department, where he is now Professor and Associate Chair. He introduced the undergraduate Software Engineering program in 1998 and was its director until 2004. He is currently developing a new masters program in Software Engineering, to be introduced in Fall 2006. As well as software engineering, his current interests include distributed computing, graphics, and artificial life.

Ruby on Rails: The Whirlwind Tour

David Heinemeier Hansson
2005 Goggle and O'Reilly's Open Source Best Hacker Award Recipient
Ruby on Rails Founder

Abstract:
Get 1st class tickets to tour with Ruby on Rails. A glimpse behind the headlines, a look at the fundamental shift that this Yet Another Framework is bringing to the world of web-application development. See the beautiful of domain-specific languages in full effect and learn about the holy cows we had to slaughter to enjoy the view.

Biography:
A product of Danish Design from the Winter of '79. Grew up, graduated, and still live in the city of Copenhagen. I've been writing about it all since the Reboot conference in 2001 inspired me to start Loud Thinking. Since '96, I've been working with the net with varying levels of success in the fields of game journalism, marketing, project management, design, and development. These days its mostly about development, though.

As a partner in 37signals, I helped transform the venerable design shop into a product company. Basecamp, Backpack, and Ta-da List are all applications launched since the shift came into effect in February 2004. I did the programming for all of them.
In July 2004, I released the framework Rails (also known as Ruby on Rails) from the work on these applications. I've been managing that as an open-source movement ever since. And lately, quite a few people has been taking notice. That means a bunch of speaking engagements including RubyConf, FISL, Reboot, OSCON, JAOO, and more. In August 2005, I won the Best Hacker of the Year award at OSCON from Google and O'Reilly.

In addition to Rails, I've also created the most downloaded Ruby end-user application. It's a small, light wiki called Instiki. I'm no longer actively developing on it, but still proud of how far I made it go. I even used it to write my final project towards my bachelor's degree in Business Administration and Computer Science at the Copenhagen Business School.
I believe in change, ignorance (my own), love, and the power of motivation.

Creating Passionate Users

Kathy Sierra
Co-Creator of Head First Series
Finalist for a Jolt Software Development award
Founder of Javaranch.com

Abstract:
What do game designers, neurobiologists, and filmmakers know about creating passionate users? How can we create not just user-friendly, but brain-friendly software? How can we help inspire users at a deeper emotional level?
By reverse-engineering passion, we learn the key attributes shared by the things people are passionate about. And if we can figure out how to incorporate some of these attributes into our software, APIs, and documentation, we can create applications that can inspire users to love and ultimately evangelize what you create. Thanks to the latest research in brain science, we now have a much clearer path for creating user experiences that can turn even the most mundane task into an engaging interaction.
Whether you're building commercial applications, developer APIs and frameworks, or end-user documentation and training, user/brain-friendliness can make the difference between frustrated users and those who can't wait to see what you come up with next.

Biography:
Kathy Sierra has been interested in the brain and artificial intelligence since her days as a game developer (Virgin, Amblin', MGM). She is the co-creator of O'Reilly's bestselling Head First computer book series (winner of the Jolt Cola/Software Development Magazine award in 2004, and named to the Amazon Top Ten Computer Books of the year for the past two years). She's also the founder of one of the largest programming community web sites, javaranch.com. A former master trainer for Sun Microsystems, she spent several years teaching engineers lthe latest Java technologies. Most recently, she's been writing the "Creating Passionate Users" blog and book (published by O'Reilly in early 2006).

Fight the Traffic

Chad Fowler
Author of
My Job Went To India
(And All I Got Was This Lousy Book)

Abstract:
Despite what you may have heard on the news, it's a great time to be a software developer. Opportunities abound, but most of us just aren't looking for them. The sky is falling, but it's nothing to complain about. We'll take the tumultuous environment of global software engineering and turn it into a playground for the passionate programmer. When is Starbucks better than your locally owned favorite? What can we learn from Wal-Mart? Why was Apple stupid enough to get into a commodity market like MP3 players?

Biography:
Chad Fowler has been a software developer and manager for some of the world's largest corporations. He recently lived and worked in India, setting up and leading an offshore software development center. He is cofounder of Ruby Central, Inc., a non-profit corporation responsible for the annual International Ruby Conference, and is a leading contributor in the Ruby community. Chad is a contributor and editor for numerous books and is author of the recently released, My Job Went to India (and all I got was this lousy book): 52 Ways to Save Your Job and the upcoming Rails Recipes.

A Panacea or Academic Poppycock: Formal Methods Revisited

Connie Heitmeyer
Head of the Software Engineering Section of the Naval Research Laboratory's Center for High Assurance Computer Systems, Chief Designer of the SCR

Abstract:
Most programmers avoid formal methods and their support tools due to the perceived difficulty of applying them. This talk describes the many different roles that formally based tools can play in debugging, verifying, and validating software and software artifacts, with emphasis on tools for specifying and analyzing software requirements.

Tools for requirements construction and analysis are of special interest because capturing and documenting requirements presents one of the most difficult problems in software development. The talk also describes the presenter's recent experience and lessons learned in specifying software components of NASA's International Space System and in the formal specification and verification of a security-critical cryptographic system. The talk concludes by identifying some open problems in software engineering that require new research.

Biography:
Connie Heitmeyer is the chief designer of the SCR (Software Cost Reduction) toolset, a formally based set of tools which has been distributed to more than 200 organizations in academia, industry, and government and applied to many real-world systems. The head of the Software Engineering Section of the Naval Research Laboratory's Center for High Assurance Computer Systems, she recently served as co-program chair for MEMOCODE 2005, the 3rd International Conference on Formal Methods in Hardware/Software Co-Design, and as co-chair of the 2005 Experience Reports Track at the International Conference on Software Engineering. She is a member of the editorial boards of the ACM Transactions on Software Engineering and Methodology, the Requirements Engineering Journal, and the Journal on Software and System Modeling. Her research interests are in formal specification and formal analysis of software and system requirements and of high assurance software systems. She is also very interested in transferring formal methods technology and tools to software practitioners.

Presentations, Tutorials, and Papers

I
Academic Presentations

Recursive Software Engineering for Tomorrow's Software Engineers

Nancy Acemian
Concordia University, Montreal, Canada

Abstract:
How can computers and software be used better to teach software engineers? By having software engineers develop adjunct learning and teaching environments for each other to complement conventional in-class learning. Tools such as lecture videos coupled with annotated Power Point slides (Video Streaming/Flash), Java applets illustrating program segments and randomly generated on-line exercises (PHP/MySQL) are some of the learning tools available off the web to students of an Object Oriented Programming course at Concordia. The environment was developed by Concordia SOEN students and continues to be maintained by one of the original student developers. This presentation will describe the project, the development, and the role of Software Engineering in this in-house project.

Biography:
Nancy Acemian teaches programming in the Department of Computer Science and Software Engineering at Concordia University since 2000. Previous to this appointment, she taught Computer Science at Marianopolis College in Montreal for 11 years. She holds a BA of Commerce with a Major in Mathematics from McGill University, Montreal, and an MA in Computer Science from Concordia University where she is also pursuing a PhD in Educational Technology. Her research area is in the visualization of code, to aid students "see" the sequence of programming codes, and to develop better learning outcomes. Another objective is to produce effective learning tools for different learner styles which can be used in class and on-line. Nancy Acemian is also assists the Concordia University Centre for Teaching and Learning Services (CTLS) facilitate workshops and seminars on teaching for faculty and PhD students.

Model-Based Development of Advanced User Interfaces: Integration of Audio-visual Interaction and Task Specification

Peter Forbrig
Rostock University, Rostock, Germany

Abstract:
The use of techniques from the fields of visualization, natural language and task modeling provides a new complementary style of human computer interaction, where the computer becomes an intelligent, active and personalized collaborator. In this talk we present an adaptive, platform independent integration strategy of appropriate state-of-the-art visualization, speech and task modeling techniques with a special focus on interfaces for mobile devices. Different XML-based languages are used for this purpose. The talk will also address the potential synergy among several interaction technologies and how they can be combined together to build a new generation of human-computer interfaces. The implemented system is illustrated using an automated maintenance support case study.

Biography:
Dr. Forbrig is a full professor of software engineering at the University of Rostock in Germany. He got his PhD in compiler construction (1980) and his habilitation in software engineering methods (1987). Besides working in industry from 1981 to 1998 he was appointed as a full professor in 1994. His research interests include classical software engineering like UML, design patterns and case tools. Additionally, he is interested in combining task-based development methods with object-oriented once. His research combines human computer interaction with software engineering. Dr. Forbrig published several papers on software engineering and HCI. He published several textbooks in German and he is the author of a German UML book.

Dr. Forbrig is vice chair of IFIP TC 13.2 and was visiting professor at the University of Cottbus (Germany, 1993), University of Linz (Austria, 1997), University of Potchefstroom (South Africa, 2000) and Concordia University Montreal (Canada, 2003).

Software Testing as a Social Science

Cem Kaner
Florida Institute of Technology, Florida, USA

Abstract:

Software development groups spend hugely on testing. Some companies (e.g. Microsoft) assign equal numbers of testers and programmers to projects. Despite the large role on real projects, the place of testing in the computer science or software engineering curriculum is usually trivial. Suppose we added more testing and training to SE students. What should we add?

One exciting vision brings testing closer to the programming mainstream. I like this vision. It leads to cleaner code, better test tools, skilled test architects, friendly project team dynamics.
Consider another vision. Imagine a computer program as a communication among people and machines, distributed in space and time. Programming focuses on communication between the person and the machine. What about the person-to-person issues? I like Jerry Weinberg's definition of quality: Quality is value to some person. Along with the pragmatism (greater quality if and only if higher value), it highlights the subjectivity of quality. Different stakeholders, different values, different quality.

When we search for clues to better and more relevant testing in the needs, preferences, valuation and conflicts among stakeholders, in complaint patterns and market reactions to our previous products and our competitors', when we use human performance measures as indicators of project status and product quality, when we use intuition or formal tools to find patterns in the overwhelming mass of conflicting information about the products we are testing, we are applying the social sciences, not programming.

Imagine the tester as an investigator, someone who uses psychological/ economic/ anthropological/ forensic tools and insights to expose quality-related information about the product under test. What would her job look like? What would distinguish strong work from weak? What should she study, what might we teach to help her along?

Biography:

Cem Kaner is a Professor of Software Engineering at the Florida Institute of Technology and an attorney focused on the law of software quality. He is senior author of several books, including Testing Computer Software, and of online open courseware at www.testingeducation.org. His undergraduate and doctoral studies were at Brock, McMaster, York and Windsor.

Empirical Studies in Software Engineering: A step closer to Usefulness

Ahmed Seffah
Concordia University, Montreal, Canada

Abstract:
Software testing has always been one of the main pillars of software engineering. Testing comes in several flavors, from testing lines of codes and correctness of algorithms, all the way to the correctness and completeness of requirements. In this talk, we focus on testing software with people, namely end users, to make sure the application will meet their needs and will work correctly in its context. Empirical testing can greatly contribute to user acceptance of the software while reducing development time and training costs.

Biography:
Ahmed Seffah's interests are at the intersection of human-computer interaction and software engineering, with an emphasis on human-centered software engineering, empirical studies, theoretical models for quality in use measurement, as well as patterns as a vehicle for capturing and incorporating empirically valid design practices in software engineering practices. He is a co-founder of the Usability and Empirical Studies Lab and the founder and chair of the Human-Centered Software Engineering Research Group at Concordia University.

II
Corporate Presentations

A Software Developer's Experience with Agile

Patrick Ng, Aditya Thakur, Gang (George) Zhu
Motorla

Abstract:
The best known and oldest software development lifecycle is the waterfall model, where developers follow the steps of requirements gathering, analysis, design, coding and testing in order. Recently, a project team at the Motorola Global Software Group Canada centre in Montreal experimented with an "Agile" process for the development of a new software feature. The development itself presented a number of challenges, such as the significant enhancement of the software functionality, an aggressive schedule, and a number of technical unknowns. As part of the project, the team utilized the following Agile techniques: prototyping, just enough documentation, pair programming, iterative and incremental development, refactoring, constant integration, constant communication, test first development and automated testing. In the end, the team observed increased productivity, boosted creativity, and lower cost of quality. In this presentation we will investigate the reasons behind these improvements as well as share the team's experience in practicing Agile, as compared to traditional, non-Agile, methods.

Eclipse Performance

Chris Laffra
Rational Performance Engineering Team,
IBM

Abstract:

To scale the Eclipse platform to a large product, plug in developers will t some point have to study both their CPU performance and their memory consumption. Eclipse offers non-traditional performance challenges that have to do with the adoption of a large framework. It has been wisely coined that every Computer Science problem can be solved by adding one more layer of abstraction.

The various Eclipse abstractions such as plugins, extension points, and features, allow developers to grow Eclipse to an unprecedented size. Some products include over 2,000 plugins. However, profiling tools typically only show low-level details that make it hard to rediscover the abstractions. For instance, one single API call in JDT may unwittingly result in millions of method calls if the workspace is large. Stack traces of 500-600 deep are not exceptional.

How do we profile such large applications and make sure we don't get lost? In addition to performance, memory consumption is highly relevant for Eclipse applications. Most Eclipse applications, including IDE extensions, are just in the business of converting data into different formats, such as from XML into a binary registry, Java source into class files, JSPs into Java and HTML, etc.

Data conversion is often an expensive process, and plugin authors quickly resort to using a cache to play the space/time tradeoff game. Monitoring Java heap growth and doing blame analysis is far from trivial. It can be quite difficult to discover who owns a certain string when the heap measures 600MB and contains millions of objects. What we will go over in this session is a set of publicly available profiling tools, and see how they can be used to profile Eclipse and analyze its heap usage and detect leaks. Various live demos will be given on real Eclipse scenarios and we will see how these profiling tools help address complexity. We will show how certain design decisions influence how we can trace activity at runtime, and how profiling tools can be enhanced to benefit better from them.

Biography:

Chris Laffra was born in The Netherlands and obtained his MsC at the Vrije Universiteit of Amsterdam in 1988 and a PhD at the Erasmus University of Rotterdam in 1992. At both IBM T.J. Watson Research Center and Morgan-Stanley, Chris worked on tools for user interfaces, component infrastructures, program analysis, debugging, visualization, compression, and

optimization. He led the OTI Amsterdam lab for 3.5 years, working on WebSphere Studio Device Developer®. At IBM Canada's lab in Ottawa he worked on the border between Java™ runtime environments and Eclipse (and co-authored The Official Eclipse 3.0 FAQs). Currently, Chris works at IBM RTP to improve RAD/RSA performance.

Agile–Helping you Deliver Useful Software

François Beauregard
Research and Development,
Pyxis Technologies

Abstract:
The software development industry has a very bad track record in delivering useful software to organizations. In its widely referred research 'The CHAOS Report (1994)', the Standish Group has found that on successful projects, 45% of the functionalities developed are never used and another 19% are rarely used. Therefore, the potential for improvement is huge.

During this presentation, we will identify some potential causes of such poor performance and then explore how Agile software development methodologies can help deliver more useful software. Topics such as requirements gathering, incremental funding method (IFM), project metrics and collaboration between project stakeholders and development teams will be discussed.

OSGi: The Open Source and Standard Platform of Choice for Restrained Devices

Louenas Hamdi
Researcher, SAP Labs Canada
SAP

Abstract:

OSGi offers a component/service oriented computing environment for networked services. Enabling a networked device with an OSGi framework adds the capability to manage the life cycle of the software components in the device from anywhere in the network without ever having to disrupt the operation of the device. Software components are libraries or applications called bundles that can dynamically discover and use other components thru service sharing mechanism.

OSGi offers many standard component interfaces that are available for common functions like configuration, device access manager, universal plug and play, wire admin and many more. The OSGi specifications are broadly applicable in many areas, and especially to restrained environments, because it is a thin standard layer that allows multiple components to efficiently and securely cooperate in a single Java virtual machine. Unlike other Java technologies like JMX or MIDP, the OSGi service platform allows bundles to supply code as well as services to the environment. In restrained environments sharing code is important because it allows libraries with shared functionality to be exposed to all the allowed applications and therefore reduce the code redundancy and the applications size.

The presentation will illustrate a step by step application-building example and will show some very practical and real life examples using OSGi.

Biography:

Louenas Hamdi joined the SAP Research Team in Montreal on January 1st, 2004. He is currently involved in different projects within the SAP Smart Items Research Program. Louenas received his Master degree in Software Engineering from ETS (Ecole de Technologie Supérieure), Montreal, Canada. He also holds an Engineering diploma in Computer Science from Université de Tizi-Ouzou, Algeria. His research interests are in the domains of: Smart Items, Mobility, Enterprise Applications, Context Awareness, RFID, OSGi.

Software Engineering in the Video Game Industry

Alex Hyder
Development Director – NHL ’07
Electronic Arts Montreal

Abstract:

Modern software engineering has its roots in the military world, and even today is most rigorously applied to the development of mission-critical systems in the aerospace and telecom industries. The games industry has traditionally followed an extremely informal approach to software development, driven by the need for creativity and speed of development. However while, many of the drivers for process formalism such as schedule predictability, quality, and cost of rework apply to game development, others, such as requirements traceability and lifecycle costs do not. As a result, there is some debate among game developers as to what kind of software engineering practices should be followed in the game industry. This presentation is a result of the author’s personal transition from the formal practices used in the manned space program to the fast and flexible world of game development. It will examine some of the specific challenges of game development, and will discuss some of software engineering practices currently being adopted at Electronic Arts.

Biography:

Alex completed his B. Eng (1983) and M. Eng (1989) at McGill University in mechanical engineering, specializing in robotics. For the next 5 years, he worked as a software developer at the NASA Johnson Space Center, developing software simulation tools for the analysis of spacecraft dynamics and robotic systems as part of the Space Shuttle mission planning process. That was followed by several years in the telecom industry, first as a real-time software developer at Nortel, then as a project manager at Motorola’s Montreal Software Center. Alex has worked in the game industry since 2002, managing game teams developing for the Playstation2, Xbox, GameCube, PSP, and PC. His most recent projects include Medal of Honor European Assault and SSX On Tour. He is currently responsible for NHL ’07 PS2, Xbox, PC and PSP.

Testing in a Creative Environment

Karine Roy
SQA manager
Autodesk Media & Entertainment

Abstract:

Many people think of testing jobs as an entry level position and a step to move towards the job their really want to do. Many will think anybody can become a tester, as it is just a matter of knowing the product under test. Many believe testers should not be involved in the development life cycle, or should only be involved at the end. Many engineers see testers as a necessary pain, but have very little desire to develop a closer relationship with them, as they don't see the value. Many believe we should invest in development more than we invest in testing. Are those beliefs true, what if they weren't????

In this presentation Karine will talk about those beliefs that surround testers in their day to day lives, and how overcoming those beliefs could lead to positive results.

Biography:

Working in the Film, Video and 3D industry since 1995, Karine has held a variety of software testing positions ranging from core tester, automation tester, automation lead and QA team lead. For the last few years she's been working as a QA manager at Softimage Avid and, since 2004, at Autodesk, in their Media and Entertainment division. Karine is leading a team of over 60 test specialists, the majority of which being full time employees. She also manages remote team in India.

Autodesk's M&E products range from 3D applications to 2D color grading, effects and editing as well as video compression and encoding. The various projects Karine has been involved in create and deliver authoring tools for creative professionals making Computer Generated Imagery (CGI) for feature films, commercials and video games.

III
Tutorials

Considering Ajax

Chris Laffra
Rational Performance Engineering Team,
IBM

Abstract:

Lately, there is a lot of interested in Ajax (Asynchronous JavaScript plus XML).Various Ajax applications demonstrate a much more interactive rich client experience than traditional web browsing. Using Ajax, new and innovative aggregation and presentation techniques can be deployed in an unprecedented fashion. Inspired by Alex Bosworth's list of Ajax mistakes, I compiled a list of 20 attention areas to look at when considering Ajax techniques for a website. Some of them are potential problem areas, such as breaking the "back" button, causing unhappy off-line experiences, not showing progress, loosing bookmarkability, and raising security concerns. Most of them, however, are indicating the high potential Ajax has. Of course, the presentation itself will be done in a browser, using Ajax techniques as much as possible. The talk can be found at http://eclipsefaq.org/chris/ajax. I have experimented a lot with Ajax in the past, doing things such as enhancing Google maps to find a new home in Raleigh near a good school, and generating the online version of the Eclipse FAQs at http://www.eclipsefaq.org/chris/faq/indexb.html.

Biography:

Chris Laffra was born in The Netherlands and obtained his MsC at the Vrije Universiteit of Amsterdam in 1988 and a PhD at the Erasmus University of Rotterdam in 1992. At both IBM T.J. Watson Research Center and Morgan-Stanley, Chris worked on tools for user interfaces, component infrastructures, program analysis, debugging, visualization, compression, and optimization. He led the OTI Amsterdam lab for 3.5 years, working on WebSphere Studio Device Developer®. At IBM Canada's lab in Ottawa he worked on the border between Java™ runtime environments and Eclipse (and co-authored The Official Eclipse 3.0 FAQs). Currently, Chris works at IBM RTP to improve RAD/RSA performance.

Fundamental Issues in Software Metrics

Cem Kaner
Florida Institute of Technology, Florida, USA

Abstract:

What are we measuring when we collect and compute software metrics? What gives us confidence in these measurements? What are the risks of taking them or using them? Measurement is very important for software projects. How else can we tell whether we are likely to meet a schedule, ship an acceptable product, overrun the budget? How else can we tell that this programmer needs help and that programmer is far enough ahead to provide it? What else could provide a basis for estimating the size and difficulty of a project and plan the staffing, schedule and cost accordingly?

Measurement is very important for software projects. And so, when I say that many of the popular metrics in use today have little theoretical basis, that software engineering is decades behind other fields in its application of basic measurement theory, and that measurement programs are probably so rare in industry because so many have been abandoned after doing more harm than good—some people respond the way they'd respond to someone who argues that because doctors sometimes commit malpractice, you should never seek medical services.

That's not what this talk is about. I'm not slamming medicine. I'm saying, "Don't buy snake oil. Or if you do, don't expect it to cure what ails you." I'm not slamming metrics. I'm saying, "Don't use unsound measures. Or if you do, use them with skepticism and great care. And work on creating and validating some replacements."

This tutorial throws down a challenge to students who are preparing to apply software engineering on the job or research it in their dissertations. You /will/ take and report measurements. You'll have to. The question is, will you know enough about the measures you use to be credible and add value.

Biography:

Cem Kaner is a Professor of Software Engineering at the Florida Institute of Technology and an attorney focused on the law of software quality. He is senior author of several books, including Testing Computer Software, and of online open courseware at www.testingeducation.org. He undergraduate and doctoral studies were at Brock, McMaster, York and Windsor.

Software Start-up

Laurent Seiter

Abstract:

What is a startup ? Why start one ? What are the basic ingredients to make it successful, or a failure ? All startups are not the fairytales of the Internet Bubble and everybody does not become Google overnight.

This tutorial coming from a real-life experience will offer hints and directions to answer the above questions. We will travel through the different layers of the cake, from the original idea to the motivating impulse, the context, the lifestyle, the business plan, the funding, the morale, the marketing and PR, the recruitment, how to deal with customers, bank managers, incubators, investors and associates, and other delights of the startup experience. Attendees with a project will come out with a clearer view of what to expect from a software engineer point of view: the tutorial is closer to a report from the field than to a theoretical MBA course.

Biography:

Laurent Seiter has 14 years of experience in software development in several industries (telecoms, logistics, stock exchange, groupware) and research labs (CERN, CRIM). He has been the co-founder of a software startup in 2000 and has created other entities in the music industry.

Static Analysis Using the Eclipse Test and Performance Tools Platform (TPTP)

Orlando Marquez

Abstract:
This talk introduces the static analysis framework and code review components built into the Eclipse Test and Performance Tools Platform (TPTP). The static analysis framework offers users a consistent interface through which all forms of analysis can be manipulated. For the developer, TPTP supplies a simple API for creating analysis providers, developing rules and presenting analysis results to the user. This session will initially provide a quick walk-through of the new user interface components including a discussion of the design considerations that help improve the user experience when analyzing resources in the Eclipse workspace. This overview will demonstrate the Java and C/C++ code review providers used to analyze sample source code to generate and view results. Focus will then shift to an introduction of the supplied API and will include two examples to illustrate the steps needed to integrate an existing third-party analysis tool and to create new rule provider from first principles. This will include detailed information describing which extension points are available, how to define rule categories, rules, results and viewers. Following the provider discussion, the focus will shift to the Java code review provider supplied in TPTP. Though this provider supplies approximately 70 rules for common J2SE issues, it also offers developers a trivial API for augmenting the rule set with new custom rules. Examples taken directly from the open source TPTP code review rules will quickly walk developers through some basic JDT-based API's available for Java rule creation. This knowledge will then be used to write a simple rule that can be plugged into the TPTP Java coded review engine. Finally, this tutorial will describe some of the more advanced features of rule writing such as rule templates, variables, detail providers and quick fix support.

Biography:
Orlando Marquez is a Software Engineering student from the University of Waterloo. As part of his last internship, he worked at the IBM Ottawa Lab developing Application Analysis features for Rational Software Architect and related products. He also contributed greatly to the Eclipse Test and Performance Tool Platform (TPTP) Static Analysis tooling. His current main areas of interest are source code and runtime analysis.

IV
Papers

Task Models and Remote Usability Testing

Gregor Buchholz, Peter Forbrig, Anke Dittmar, Andreas Wolff, Daniel Reichart
University of Rostock
Department of Computer Science
Albert-Einstein-Str. 21
18051 Rostock, Germany
Tel: +49 381 4987624
grbuc@informatik.uni-rostock.de

ABSTRACT
This paper discusses the integration of remote usability testing into a model-based approach of software development. The development process consists of a sequence of interactive model transformations. It is shown how first prototypes of interactive systems, which in our approach are animated models or interactively generated applications can help to capture requirements and how the models evolve to the final interactive system. We also demonstrate how to enable usability experts to use this model-based approach for testing the usability of software in early development stages based on the tasks users have to perform. A tool is presented, which visualizes the activities of a test user based on the models. The tool supports remote usability tests, which even can be performed on mobile devices.

Categories and Subject Descriptors
D.2.1 Requirements/Specifications

General Terms
Languages

Keywords
Model-based Design, Remote Usability Tests, Patterns

1. INTRODUCTION
To meet the purpose a software system is intended for is a main challenge in software development and it is commonly accepted that the development process has to start with the analysis of the problem domain users work in. There are some discussions whether one has to start with analyzing objects, tasks, or interactions but at the end there is a common understanding of the importance of all aspects of the problem domain. It is also more and more accepted that the users' view is most important for the software under development. A user-centered development process perfectly supports this idea.

Model-based development of software systems has become more and more popular. Even if it is up to now not used very extensively it is an attractive process with proven record of success, especially in the context of developing multiple user interfaces. There are approaches focusing first on object models like the model-driven architecture of UML [24]. However, we follow task-based approaches like ADEPT [26], CTTE [4] or Cameleon [1]. Typically, such systems are used to model existing or envisioned tasks. They help to understand the tasks a user has to perform in more detail by allowing simulations. Additionally, systems like TERESA [22] support the development of user interfaces.

Our System DiaTask [19] follows a similar approach. Based on task-, object-, user- and environment-models, interactive systems are developed. The next chapter will describe this approach in further details. Afterwards opportunities for remote usability tests will be sketched and at the end we will discuss reached and further goals.

2. MODEL-BASED DEVELOPMENT
We strongly believe that software engineers and user interface designers have to base their work on the same models. In Figure 1 these models are depicted on the left hand side. Here the device model is a representative of a general environment model. These models are as well the basis for the development of the soft-ware developed by software engineers as those for the software of user-interface experts.

Furthermore, we consider software development as a sequence of transformations of models that is not performed in a fully automated way but by humans using interactive tools.

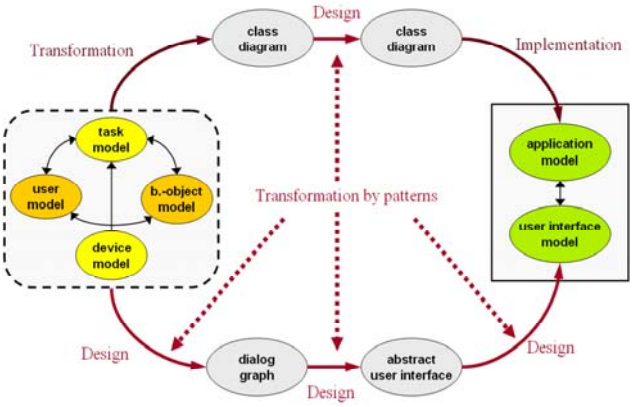

Figure 1, Model-based development process model

Our work is especially focused on methods and tools supporting transformations by patterns. The transformation of class diagrams by patterns using Rational Rose is described in [11]. In [20], the idea of supporting the development of task models by patterns is shown.

In the following we demonstrate the application of our ideas to a small example of developing a mail management system. Figure 2 is the result of the interviews with forthcoming users. It demonstrates how task models look like.

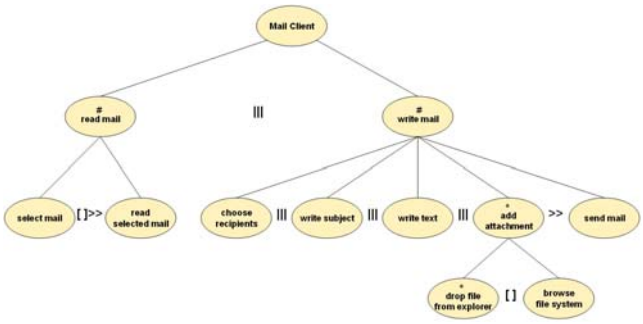

Figure 2, Task model for the mail management system

According to the task model of Figure 2, a user may either read his mail, or write a new one. To read his mails, he has to select a specific mail from a list that is generated and presented to him by the application. Once he has selected a mail its content is displayed. Select and display are consecutive subtasks of an iterative tasks that can be cancelled at any time.

Writing mails is modeled in a similar manner. After a user decides to write a mail he has to enter the iterative task produce mail, where he is requested to compose a new mail and, after having finished this, the application sends it away. This sub-task may also be cancelled at any time.

In addition to the classical temporal operations like >> - enabling, ||| - in parallel, [8] – alternative a new operation symbol # is introduced. It represents the "instance iteration" operation. In contrast to the classical iteration * it allows to start a new iteration before the old one is finished. Thus, it is a specification feature, which is very helpful in a lot of applications.

With one of our tools DiaTask [19] we are able to develop a dialog graph that represents the navigation structure of the interactive systems. Such a graph is based on the previous specified task model.

A dialog graph consists of a set of nodes, which are called *views* and a set of transitions. There are five types of views: single, multi, modal, complex, and end views. A single view is an abstraction of a single sub-dialog of the user interface that has to be described. A multi view serves to specify a set of similar sub-dialogs. A modal view specifies a sub-dialog, which has to be finished in order to continue other sub-dialogs of the system. Complex views allow a hierarchical description of a user interface model. End views are final points in (sub-) dialogs. Each view is characterized by a set of (navigational) elements. A transition is a directed relation between an element of a view and a view. Transitions reflect navigational aspects of user interfaces. It is distinguished between sequential and concurrent transitions. A sequential transition from view v1 to view v2 closes the sub-dialog described by v1 and activates the sub-dialog, which corresponds to v2. In contrast, v1 remains open while v2 is activated if v1 and v2 are connected by a concurrent transition. Figure 3 shows the graphical notation for the different types of views and transitions.

Unlike to TERESA [22] the dialog graph is the result of a design process and not the result of automatic transformations. DiaTask allows to assign several different dialog graphs to one task model. Figure 3 demonstrates the example our mailing system with single views (main window "Mail Client", write mail), multiple views (read mail) and the end view (end). The screenshot is produced using the eclipse [9] plug in for DiaTask.

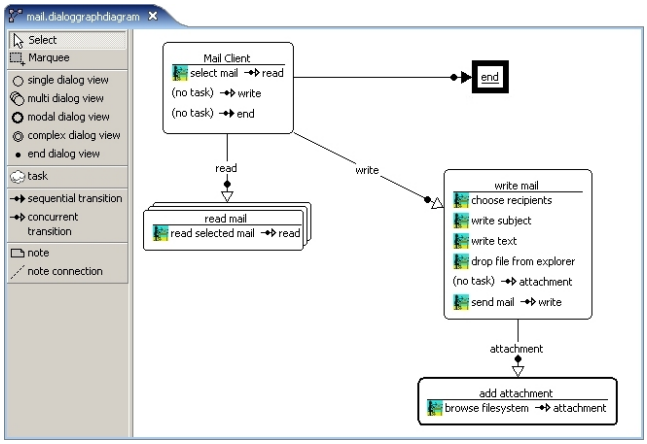

Figure 3, Dialog graph for the management system of mails

There are concurrent transitions from "mail client" to "read mail" and "write mail". This means that "main window" stays open and can be activated by a mouse click.

Multiple views are able to instantiate several instances. In the example of a mail system this means that users can read several mails in parallel. Multiple views go together with concurrent transitions. Figure 4 illustrates the abstract prototype generated from the model of figure 3. It captures a situation where one mail is read and another one is written. The "main window" is active and as well pressing the corresponding button can activate end, read mail or write mail.

Figure 4 demonstrates an abstract user interface of the mail client application.

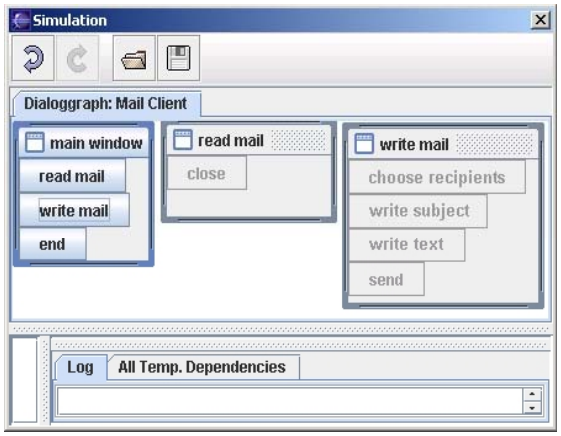

Figure 4, Canonical abstract prototype of Fig. 3 in animated mode

The abstract user interface shown in Figure 4 is automatically generated from the specification of the dialog graph and behaves according to the temporal relations defined in the task model. It is already a very good instrument to improve the communication with users during the requirements analysis phase. Unfortunately it has the drawback of a very abstract interface. We intended to improve this situation. It was our idea to generate the abstract user interface (e.g. Fig. 4) in a language which can be picked up by a GUI-editor for improvements. We decided to use XUL [32] for

this purpose. This user-interface description language was introduced with the Mozilla project [15] and part of the success story. Based on an existing GUI plug in for eclipse an editor for XUL was developed. This editor is able to replace existing elements by other ones. In this way, the abstract user interface can be improved to a more useful one while keeping up the reference from the GUI elements to tasks. We are especially working on the problem of how patterns can be used for this purpose. Possible tool support is discussed in [20].

The interpreter of the models, which controls the animation, recognizes the existence of improved windows and includes them into the animation process. In this way, the user is able to have a look at a user interface, which is already a candidate for the final interactive system.

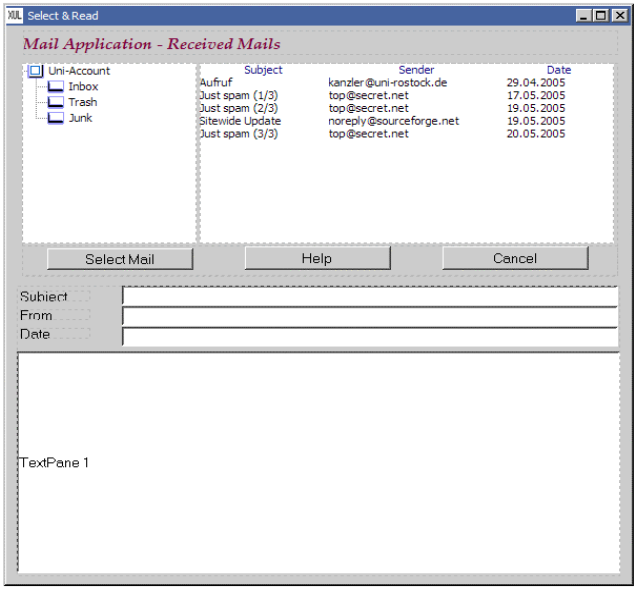

Figure 5, Designed GUI for Select & Read

Details of the GUI-editor can be found in [28] and [29]. During simulation our system offers a view on the prototype of the user interface and a view on the animated task model. Figure 6 demonstrates how the animated task model is visualized. Basic tasks (represented by squares) with green circles can be executed. Red crosses represent a status of the task that allows no execution because of restrictions (e.g. temporal relation between tasks). A blue tick signals the successful execution of a task.

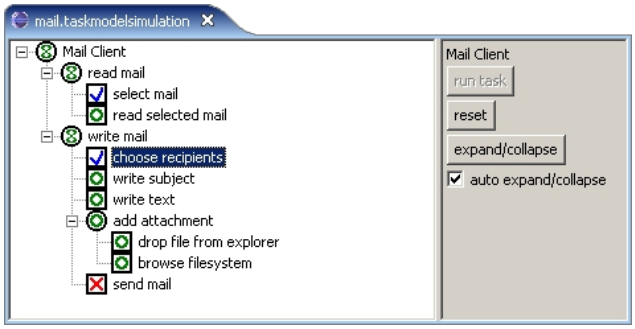

Figure 6, Visualization of an animated task model

3. REMOTE USABILITY TESTING

Testing and improving the usability of software is a time consuming process. Test scenarios have to be developed, test users have to be hired and experienced usability experts have to observe the behavior of the test user. Sometimes it is very difficult to have test user and the usability expert as test supervisor at the same time at the same place. This holds especially true for mobile applications.

Furthermore, users may behave differently depending on whether they are working in a traditional usability laboratory or in their usual environment. Sometimes only the presence of the test supervisors influences the test users by executing their tasks.

These problems can be reduced by remote usability tests. This kind of test allows the test user and the test supervisor to work at different places and even at different times. This is not new but based on our model-based development approach new opportunities are created.

3.1 Software Architecture

It is possible to run a model-based system on a client-server architecture. In this way, models are interpreted on a sever and the results are delivered to the clients. Figure 7 gives an impression of how this architecture looks like.

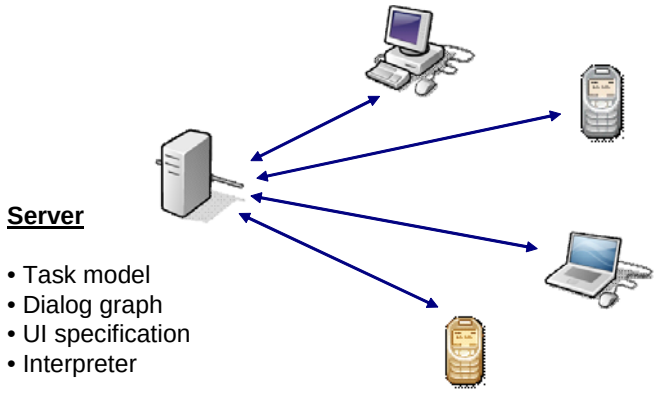

Figure 7, Software architecture

The architecture of Figure 7 creates new opportunities for remote usability test because in addition to videos and the capturing of screens the status of the interpreted models can be observed. Some features of our TMSClient and TMSServer (TMS = Task Model System) will be discussed within the following paragraph.

3.2 Tool Support for Remote Usability Tests

Based on our model-based approach, a TMSServer was implemented which allows remote usability tests in early software development stages. The server is able to interpret models, to receive state change events from the client the user is working on and to send the results to the client (see Figure 8).

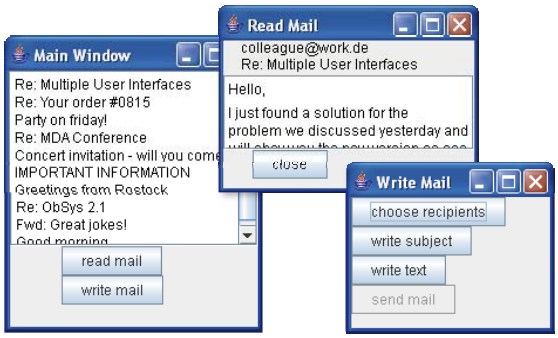

Figure 8, User interface for the test user

The usability expert uses another client software that uses the task state change notifications to give an impression of the test by visualizing the actual state of the execution of the task-model instance, which is presented in Figure 9. We do not want to comment to all the information presented but would like to draw the attention of the reader to the left side of the screen shot.

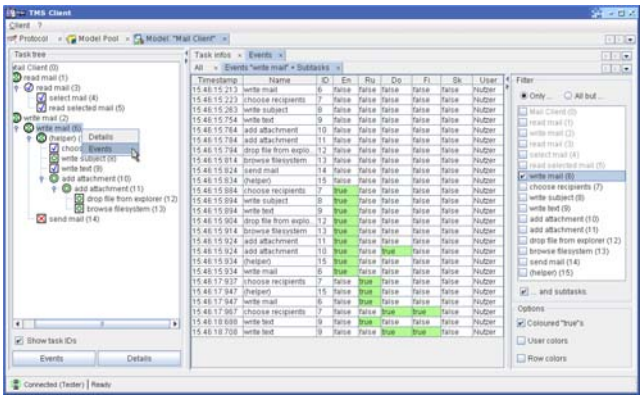

Figure 9, User interface (animated task model) for the usability expert

The user interface of Figure 9 was technically produced based on a Java implementation, where parameterized cascaded observers were used to observe the states of the models on the server.

At the moment he usability expert has to watch the changes of the task model instance. He has to observe whether the test user behaves as expected or whether tasks are activated, which have nothing to do with the actual test scenario. In the latter case something is wrong with the system and a usability problem might be found. The table in the center lists the state change events of the model's subtasks. Currently, there are five states a task can have: enabled, running, done, finished and skipped. Analyzing these state changes is intended to assist the usability expert in finding problem situations that occur during a test. In the future there will be further tool support.

One can imagine that predefined scenarios are stored and as long as the test user acts according to these scenarios nothing happens. Otherwise the usability expert is informed that something has happened that was not expected. It is his decision what to do next.

Another idea is to compare the data collected during the tests of different user interfaces for the same task model. Thereby, the efficiency of the interfaces can be measured and optimized.

Of course, it is not the intention to replace all other kinds of observations by this method but it is intended to use this concept additionally.

4. SUMMARY AND OUTLOOK

Within this paper the integration of usability tests into model-based development (Figure 1) process was proposed. This process model postulates the idea of having the same models as basis of the work of software engineers and usability engineers. This fact is very important for us. Even if not every detail of the models is important for both groups of the different developers there have to be common models as a basis of the development process. This is one big first step towards bridging the gap between software engineering and usability engineering. The approach has the following advantages:

- Software engineers are focused on the tasks a user has to perform, on the object he has to manipulate and on the work situation (context of work). In this way it is guaranteed that the developed interactive systems are user-centered.
- Usability experts are involved in the specification of models in a very early development stages. In this way they can influence the implementation process of the software engineers.
- Usability evaluation is supported during all stages of the development process based on the existing models. Especially remote usability testing can be supported.

We have been using our approach in different small projects. However, up to now we did not use models for projects of large scale. Currently we are working in a project for a mobile maintenance system together with 8 partners from university and 9 partners from industry, which follows our model-based approach. The project started in 2004 and will be finished in 2006. We are sure that the demonstrated approach will lead to a successful result of the project and that we will gain further knowledge to improve our toll for remote usability test in a mobile environment.

5. REFERENCES

[1] Cameleon: http://giove.cnuce.cnr.it/cameleon.html.

[2] Clerxkx, T.; Luyten K.; Conix, K.: *The Mapping Problem Back and Forth: Customizing Dynamic Models while preserving Consistency*, Proc. TAMODIA 2004, 33-42.

[3] Constantine L.L: *Canonical Abstract Prototypes for Abstract Visual and Interaction Design*, in Jorge J. A. et. al (Eds): *Proceedings DSV-IS 2003*, LNCS 2844, Springer Verlag, Berlin, 2003, 1-15.

[4] CTTE: The ConcurTaskTree Environment. http://giove.cnuce.cnr.it/ctte.html.

[5] Deakin, N.: XUL Tutorial. XUL Planet. 2000.

[6] Dittmar, A., Forbrig, P.: *The Influence of Improved Task Models on Dialogues*. Proc. of CADUI 2004, Madeira, 2004.

[7] Dittmar, A., Forbrig, P., Heftberger, S., Stary, C.: *Tool Support for Task Modelling – A Constructive Exploration.* Proc. EHCI-DSVIS'04, 2004.

[8] Dittmar, A., Forbrig, P., Reichart, D.: *Model-based Development of Nomadic Applications*. In *Proc. of 4th*

International Workshop on Mobile Computing, Rostock, Germany, 2003.

[9] Eclipse: http://www.eclipse.org.

[10] Elwert, T., Schlungbaum, E.: *Dialogue Graphs – A Formal and Visual Specification Technique for Dialogue Modelling*. In Siddiqi, J.I., Roast, C.R. (ed.) *Formal Aspects of the Human Computer Interface*, Springer Verlag, 1996.

[11] Forbrig, P.; Lämmel, R.; Mannhaupt, D.: *Patterns-oriented develpment with Rational Rose*, Rational Edge, Vol. 1, No. 1, 2001.

[12] Limbourg, Q., Vanderdonckt, J.: *Addressing the Mapping Problem in User Interface Design with USIXML*, Proc TAMODIA 2004, Prague, 155-164.

[13] López-Jaquero, V.; Montero, F. ; Molina, J.,P.; González, P.: *A Seamless Development Process of Adaptive User Interfaces Explicitly Based on Usability Properties*, Proc. EHCI-DSVIS'04, 2004, 372-389.

[14] Luyten, K., Clerckx, T., Coninx, K., Vanderdonckt, J.: *Derivation of a dialog model from a task model by activity chain extraction*. In Jorge, J., Nunes, N.J., e Cunha, J.F. (ed.), *Proc. of DSV-IS 2003*, LNCS 2844, Springer, 2003.

[15] Mozilla.org: *XUL Programmer's Reference 2001*.

[16] Paterno, F.; Mancini, C.; Meniconi, S: *ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models*, Proc. Interact 97, Sydney, Chapman & Hall, 1997, 362-369.

[17] Paterno, F., Santoro, C.: *One Model, Many Interfaces*. In *Proc. of the Fourth International Conference on Computer-Aided Design of User Interfaces*, Kluwer Academics Publishers, 2002, 143-154.

[18] Puerta, A.R. and Eisenstein, J. *Towards a General Computational Framework for Model-Based Interface Development Systems*. Proc. of the 4th ACM Conf. On Intelligent User Interfaces IUI'99 (Los Angeles, 5-8 January 1999). ACM Press, New York (1999), 171–178.

[19] Reichart, D.; Forbrig, P.; Dittmar, A.: *Task Models as Basis for Requirements Engineering and Software Execution*, Proc. of. Tamodia, Prague, 2004, 51-58.

[20] Sinnig, D., Gaffar, A., Reichart, D., Forbrig, P., Seffah, A.: *Patterns in Model-Based Engineering*, Proc. of CADUI 2004, Madeira, 2004.

[21] Teuber, C.; Forbrig, P.: *Modeling Patterns for Task Models*, Proc. of Tamodia 2004, 91-98.

[22] TERESA: http://giove.cnuce.cnr.it/teresa.html.

[23] UIML Tutorial, http://www.harmonia.com.

[24] UML: http://www.uml.org.

[25] UsiXML: http://www.usixml.org.

[26] Wilson, S.; Johnson, P.; Kelly, C.; Cunningham, J.; Markopoulos, P.: Beyond Hacking: A Model-Based Approach to User Interface Design, in Proc. of HCI'93, pp. 217-231.

[27] Wilson, S.; Johnson, P.: *Bridging the generation gap: From work tasks to user interface design*, In Vanderdonckt, J. (Ed.), *Proc. of CADUI 96*, Presses Universitaires de Namur, 1996, 77-94.

[28] Wolff, Andreas, Ein Konzept zur Integration von Aufgabenmodellen in das GUI-Design , Master Thesis, University of Rostock, 2004.

[29] Wolff, A.; Forbrig, P.; Dittmar, A.: Reichart, D.: *Linking GUI Elements to Tasks – Supporting an Evolutionary Design Process*, accepted for TAMODIA 2005, Gdanks.

[30] Wolff, A.; Forbrig, P.; Dittmar, A.: Reichart, D.: *Development of Interactive Systems Based on Patterns*, accepted for the workshop "Development of Interactive Systems Based on Patterns" at INTERACT 2005, Rome.

[31] XIML: http://www.ximl.org.

[32] XUL: http://www.xul.org.

How Useful are Your UML Models?

Pankaj Kamthan
Department of Computer Science and
Software Engineering
Concordia University, Montreal,
Quebec, Canada H3G 1M8
1-(514)-848-2424-3000
kamthan@cse.concordia.ca

ABSTRACT
As modeling becomes pervasive in software development, the question of the quality of resulting artifacts arises. A framework to address the pragmatic quality of UML artifacts based upon notions from semiotics, graph drawing, and cognition is presented. Feasibility of the quality goal and corresponding criteria is emphasized and the mechanisms of achieving them are identified. Examples that compromise pragmatic quality of UML artifacts are given.

Categories and Subject Descriptors
D2. Software Engineering; H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

General Terms
Measurement, Documentation, Design, Economics, Human Factors, Standardization, Languages, Verification.

Keywords
Comprehension, Software Model Quality, Pragmatics, Visual Modeling Languages.

1. INTRODUCTION AND BACKGROUND
The Unified Modeling Language (UML) [2] is a standard language of the Object Management Group (OMG) for structural and behavioral modeling in a variety of domains.

In recent years, UML has begun to play an increasingly central role in requirements and design phases of model-driven adaptive software process environments such as Extreme Programming (XP) and the Unified Process (UP). Therefore, addressing the issue of quality early is crucial from the point of view of control and prevention of problems that can propagate into later stages. If left unattended, these artifacts may, for example, fail to communicate their purpose, could be misleading to their stakeholders, or be virtually non-modifiable. This would undermine the basic philosophy of UML to unify multiple notations that were potentially threatening interoperability among tools and communicability among engineers, and could adversely affect further acceptance and growth of UML.

The previous efforts [1,6,10,18] on tackling the quality in UML artifacts suffer from one or more of the following issues: the approach is apparently not systematic, the focus is more on the solution than the problem, the coverage is limited to a specific diagram type and/or a specific quality characteristic, or the trade-offs of proposed solution(s) are not discussed.

In this paper, we address the issue of quality in UML artifacts based on the ideas from semiotics [12], and its application to information quality [11]. The work presented here is in the direction of and extends that in [8].

In semiotics, there are six levels for analyzing symbols: physical, empirical, syntactic, semantic, pragmatic, and social levels. Our focus here is on the pragmatic level, which is the practical knowledge needed to use a language for communicative purposes. We propose a framework as a first step towards understanding, assessing, and ensuring pragmatic quality of UML artifacts.

The rest of the paper is organized as follows. In Section 2, we present the architecture of the framework for addressing the pragmatic quality of UML models, and provide a detailed description of its components on decomposition. Finally, in Section 3, we conclude with challenges and avenues for future research.

2. A UML PRAGMATIC QUALITY FRAMEWORK
We adopt the following general methodology as the foundation of the framework (Table 1):

1. Identify the pragmatic quality goal;
2. Since a goal is at a too high level to be directly addressed, decompose it further into a manageable list of specific criteria (internal and external attributes of the artifact);
3. State the mechanism(s) for achieving the criteria. A mechanism can correspond to one or more criteria.

Table 1. The outline of the UML pragmatic quality framework (where the symbol <C> stands for comprehension)

Goal	**Criteria**		**Mechanisms**	
<C>	External Attributes	Internal Attributes	Assurance and Evaluation	Tools
Feasibility Analysis				

The goal-oriented decomposition is inspired by other efforts such as the Goal/Question/Metric (GQM) method [17], widely-used in organizational measurement programs. However, one of the criticisms of the GQM method is that it does not incorporate feasibility. The attribute-level decomposition is similar to that in traditional software quality models [7].

We now describe the components of the framework in detail.

2.1 Feasibility Analysis

UML artifacts are often the means to the end and their quality control carries extra cost (in form of time, effort, and resource commitment) on top of considerations for software quality. We also need to prioritize and make trade-offs among the criteria and corresponding mechanisms. Therefore, an economic, organizational, and technical feasibility analysis is necessary for a realistic realization of the quality goal. We view feasibility as a manifestation on all aspects of the framework in order to make it practical.

Further discussion of techniques for feasibility analysis are beyond the scope of this paper. We simply state any feasibility analysis ultimately requires making decisions to prioritize among the given options. To help achieve that, Decision Trees, Influence Diagrams, Conjoint Analysis, Analytical Hierarchy Process (AHP), and Quality Function Deployment (QFD), are some of the commonly used techniques.

Any feasibility analysis, however, should also be in agreement with the organizational emphasis on decision support for software engineering in general.

2.2 Goal

Pragmatics is concerned with choosing from among the given possibilities in the contextual usage of symbols to express a single meaning. In doing so, there is only one goal of pragmatic quality of a UML artifact: comprehension by stakeholders (producers and consumers, including users).

For a non-trivial artifact, it is not realistic that each stakeholder will be able to comprehend each statement made by the artifact in its entirety at all times. This motivates the adoption of *feasible* comprehension [11].

2.3 External Attributes

The external attributes are (are non-necessarily mutually exclusive) artifact properties as perceived by its stakeholder. The ones we view as relevant are: Domain-UML-Stakeholder Suitability, User Preference, Clarity, Visual Coherence, Simplicity, Familiarity, Interoperability, and Standardization.

A UML artifact created by a stakeholder addresses a (problem or solution) domain in software. Therefore, the suitability of the domain to UML and vice versa, and the stakeholder knowledge of UML, are critical. Our experience has shown that users may prefer one diagram type to the other for the sake of understanding even though they may be semantically equivalent for most purposes, like the use of UML Sequence Diagrams over Communication Diagrams. The significance of clarity (legibility of graphic or readability of text), consistency (visual coherence), and simplicity towards understanding are well-known in cognitive psychology. By taking a subjectivist epistemological position, we accept that our understanding of the world depends on our prior knowledge and experience, and therefore introduce familiarity as one of the criteria. Interoperability is necessary as same artifacts could be processed and viewed by different tools. Standardization reduces unpredictability on part of stakeholders and is known to contribute towards quality improvement [14].

2.4 Internal Attributes

The internal attributes are purely artifact-specific that impact how external attributes are perceived. They are non-necessarily mutually exclusive and improvement/detriment in one can impact the other. The ones we view as relevant are: Secondary Notation, Size, Structure, and Representation Format.

2.4.1 Secondary Notation

The secondary notation is one of the cognitive dimensions [13] and is defined as the use of layout and perceptual cues to clarify information or to give hints to the stakeholder. The UML secondary elements that affect the comprehensibility of artifacts are color, directionality, labeling, level of abstraction and refinement, morphology, positioning, typography, and white space.

2.4.2 Size

By size of a UML artifact, we mean both the area (dimensions) that a UML artifact occupies and the file size. The former will depend on the use of the secondary notation (morphology and white space). The latter will depend on area of a UML artifact and the export format being used. UML artifacts for similar projects may reuse (in whole or in part, verbatim or slightly modified) existing constructs. This external reuse of the constructs has a direct impact on familiarity for a stakeholder that may have already interacted with these constructs in the past. It will of course be important that reused constructs blend in well with the newer ones.

2.4.3 Structure

The structure of a UML artifact will depend on the use of the secondary notation, and to the extent there is internal reuse and coupling. The generalization/specialization or `<<includes>>` relationships lead to internal reuse. Low coupling is a hallmark of "good" design. We note here that low coupling will evidently also reduce the number of relationship types in a UML diagram, and thus the number of "lines" and/or "arrows", which improves readability.

2.4.4 Representation Format

In an electronic production of UML artifacts, the nature and choice of a format (text or binary) can directly impact pragmatic quality during production and subsequent transmission. Images in binary formats on magnification tend to have an incarnation of the "staircase effect" and are non-interactive. From applications of Gestalt psychology to graph drawing [5], it is known that humans more easily see smooth continuous contours than jagged ones.

Vector graphical formats serialized in the Extensible Markup Language (XML) circumvent the issues of binary formats and lend themselves to the benefits that are associated with descriptive markup such as support for metadata, legibility (at virtually any level of magnification), and sophisticated means of interaction on virtually any platform.

Scalable Vector Graphics (SVG) is a language based on XML for two-dimensional vector graphics that works across platforms, across output resolutions, across color spaces, and across a range of available bandwidths. Indeed, current UML modelers are beginning to provide support for UML serialization in SVG.

2.5 Mechanisms

The mechanisms we view as relevant are: Quality Assurance (Training in Secondary Notation, Use of Metadata, Pair Modeling, Refactoring) and Quality Evaluation (Inspections, Metrics). We have not included testing as one of the mechanisms as it is limited to Executable UML [16], which is currently not part of the standard UML.

In the following, the scope and limitations of the mechanisms are discussed for the sake of objectivity and to provide a benchmark for feasibility analysis. The mechanisms themselves are non-necessarily mutually exclusive and can indeed aid one another.

2.5.1 Training in Secondary Notation

Training in the use of the secondary notation is a necessity and would require the basics of user interface and interaction design (the mechanical and conceptual parts of artifact design, respectively) as they apply to technical diagramming.

The appropriate use of secondary notation is described below using Figure 1 as an example.

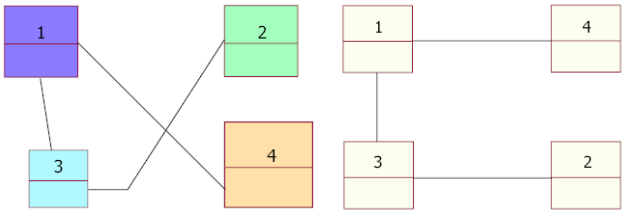

Figure 1. The pragmatic quality issues in the left UML diagram are ameliorated on the right.

- **Color.** By associating different colors with constructs in a complex figure, a stakeholder can be informed of the semantic similarity and differences between the constructs with respect to both their structure and behavior. For example, use of color in UML Class Diagrams for Java has been reported to improve the user understanding of the overall design [3]. Any use of color, however, should take into account the variations in the interpretation of primary colors by computer monitors, contrast between background and foreground, the way people with color vision deficiency view an image, and the possibility that diagrams may be printed on a black and white printer (Figure 1).
- **Directionality.** Directionality in UML constructs is critical when expressing relationships that involve the use of arrows. In cultures where English (and some others that are part of the Latin family) is the primary language of use, people usually tend to read from left to right and from top to bottom, and to ease readability long phrases and multiple lines of text need to reflect that. This is also critical in illustration of diagrams that reflect a "flow" such as the UML State Machine or Activity Diagrams. Inevitably, this also depends on the positioning of aforementioned UML constructs.
- **Labeling.** Use of application domain terminology in text labels makes it easier for non-technical stakeholders or users new to a UML extension to become familiar with the artifact. Furthermore, these labels will be more readable and reduce possibilities of misinterpretations, if they follow a natural naming [9] scheme that promotes the use of full words in preference to acronyms or abbreviations. For example, the label `ATM` has multiple expansions and therefore increases the potential for ambiguity on part of the reader as compared to for instance the label `AsynchronousTransferMode`.
- **Level of Abstraction and Refinement.** The same UML diagram could be presented at different levels of abstraction to address different needs. For example, a user-system interaction illustrated via a UML Sequence Diagram need not show the methods for system stimuli and data response during use case analysis. Not all UML constructs are appropriate for exposure to all stakeholders at all times. For example, software macro-architecture being represented using UML Package Diagrams may be much more accessible to a stakeholder interested but not directly involved in design than the UML Class Diagrams contained in it. For the sake of clarity, the well-known 7 +/- 2 organizing principle could be followed and complex diagrams could be split into multiple parts. For example, it is useful to split large Use Cases into multiple sub-Use Cases.
- **Morphology.** The morphology or shapes of nodes and vertices in a UML artifact have an impact on how the diagram as a whole is perceived by the user. It is known in graph drawing [5] that presence of crooked nodes and zigzag vertices are aesthetically unpleasing and cognitively ineffective. In general, users also associate significance with the size of nodes and vertices in a diagram, which therefore need to be consistent and semantically meaningful (Figure 1).
- **Positioning.** Humans associate positioning of graphical constructs to spatial and temporal relationships. The legibility and stakeholder interpretation of a UML artifact are affected by relative proximity of its nodes and vertices (Figure 1). Structural patterns in relationships such as symmetry and anti-symmetry are visual cues for familiarity and need to be preserved. The proper placement of a text label that is suppose to belong to one node is also important to reduce any ambiguity on part of the reader when placed between two nodes (Figure 1).
- **Typography.** The choice and the sequence of characters in the use of text (annotation and labels) affect readability. For example, the characters in a name like `O0lI1` are hard to distinguish and therefore difficult to read. The choice of fonts used for annotation and labeling depends on a variety of factors (serif versus sans-serif, amount of kerning, font size, and so forth) that are important for legibility. Fonts specifically designed for presentation on paper may in general be hard to read on a computer screen.
- **White Space.** One of the traits of any diagramming style is the introduction of white space at appropriate places. In UML artifacts, white space can be added between nodes, between nodes and vertices, and between labels and boundaries of UML constructs for clarity. Shape and positioning of nodes and vertices, and the use of white space complement each other. The use of white space can increase the file size of a model, and should be balanced with respect to the original purpose of its introduction, namely readability.

The secondary notation is the basis of several style guidelines [1] and patterns [6] specific to UML diagram types, and can serve as a basis for a checklist against which an informal quality evaluation can be carried out.

2.5.2 Use of Metadata

Metadata, such as in form of annotation, can provide further explanation on items that are not immediately obvious. It can also help capture author's intent. At times, UML constructs being used may not be common or known to the user (such as when UML extensions are being used for the first time), or the stakeholders involved may not have the necessary technical knowledge. In such cases, annotating UML artifacts using the UML Note construct can be particularly useful.

In spite of their usefulness, the author needs to be aware that the annotations do not overshadow the diagram itself (play secondary rather than primary role in the diagram), are not mere echoes of the graphical constructs, and do not contradict other annotations within or across diagrams.

2.5.3 Pair Modeling

In traditional engineering, complex, large-scale artifacts are usually crafted not by an individual but by a team. To that regard, we introduce the notion of *pair modeling* where two people participate in creating a UML artifact. Part of pair modeling can also be viewed as an informal, lightweight monitoring of UML artifacts in *real-time*: every item being drawn (or text being written) by one is under scrutiny by the partner during the creation process.

It is not automatic that any two persons put together will be more productive towards modeling than working as individuals. For pair modeling to be successful, the partners need to have compatible personality type, and a similar level of experience with the application domain and with UML itself.

2.5.4 Refactoring of UML Artifacts

UML artifacts may need to evolve for reasons such as discovery of "impurities" or obsolescence. Refactoring methods are transformations provide a systematic way of eradicating the undesirables from an artifact while preserving its semantics. In the last decade, refactoring has been extensively applied to the context of source code, and more recently to UML artifacts [15]. Figure 1 can be viewed as a result of a sequence of simple refactorings. Refactoring is beginning to have support among popular UML modelers.

However, when the UML artifacts are bound to the source code, any refactoring must also take into account any change propagation on modification of the UML artifact. Also, formally proving the invariant properties of the UML refactoring methods. remains largely unaddressed.

2.5.5 Inspections of UML Artifacts

Inspections are a rigorous form of auditing based upon peer review that, when practiced well, can help in error prevention in UML artifacts. Inspections have proved to be an effective technique in improving the overall quality of UML Class Diagrams [4].

However, inspections entails an initial cost overhead, as each participant needs to be trained in the structured review process followed by the logistics of checklists, forms, and reports involved. Moreover, the effectiveness of traditional checklist-based reading that places all defects at the same level is at times questionable. In some process maturity models such as the Capability Maturity Model (CMM), adoption of inspections amounts to achieving at least Level 3, to which a considerable number of organizations do not qualify today.

2.5.6 Metrics for UML

Metrics can provide a means for quantitative evaluation of pragmatic quality. There are metrics for UML that apply to an artifact as a whole as well as to specific diagram types and individual constructs [10]. These metrics give an assessment of structural complexity of a UML artifact, such as, the amount of internal reuse or coupling.

Still, the use of metrics faces some obstacles. Most of the metrics are introduced and used on empirical grounds, and are not formally validated against the representational theory of measurement. Manual calculations using metrics are tedious and require tool support, however, support for metrics in UML modeling tools is not currently widespread.

2.5.7 Tool Support for Automation and Modeling

UML syntax-sensitive tools or modelers can assist in successfully realizing the other mechanisms of achieving the pragmatic goal in practice. For example, a modeler may allow multiple choices with respect to font support, colors, or export formats; impose constraints on diagrams to adhere to "good" styles; or monitor changes between the model and the corresponding source code.

However, surveys have shown stark differences between commercial and non-commercial tools with respect to their ergonomics and features (conformance to official definition and versions of UML, implementation of layout algorithms, flexibility in altering properties of UML constructs, available import/export formats, support for guidelines, patterns, automatic refactoring, and metrics). This can directly or indirectly impact the pragmatic quality of a UML artifact.

3. CONCLUSION AND FUTURE WORK

Models are the "castor oil" of software engineering. UML is likely to continue playing an important role as the de facto visual language for modeling software systems. This is only underlined by the fact that the language has continually evolved in the last few years and placed within the context of meta-modeling as defined by the Meta Object Facility (MOF) which itself is a part of a higher level of abstraction of the Model Driven Architecture (MDA). Furthermore, the use of UML via its profile mechanism has entered arenas such as user interface design and ontology modeling for which it not originally designed per se.

UML artifacts are first-class citizens in software process environments that embrace them, and must strive for high quality to be useful for their target stakeholders. Addressing pragmatic quality is a stride towards that.

The framework presented in this paper provides a first step towards a structured basis for understanding and addressing the quality of UML artifacts. Before embarking on a study on quality, we must first clearly state the theoretical goal of doing so, and then for practical purposes refine the goal if necessary to make sure that it is indeed feasible. We also need to distinguish between the perception of quality from a user's perspective versus that from an author's viewpoint, and the external and internal quality attributes, respectively, are a resulting consequence.

In educational settings, such as software engineering courses that frequently make use of UML, it is crucial that the value of the quality of UML artifacts be emphasized alongside their use in software process deliverables. It is also important that this be instilled and emphasized *early* as (similar to handwriting, technical writing, or programming contexts) it becomes harder to change habits as they mature.

In conclusion, we make the following recommendations to an organization that values the production and long-term viability of UML artifacts:

- If the UML artifacts are significant in number, their production process needs to be systematically planned. This is possible by setting up a sub-process within the process of the target product.

- Along with UML training, the producers of UML artifacts need to be trained in the fundamentals of the UML secondary notation.
- It is worthwhile investing in a UML modeler that supports the mechanisms for targeting pragmatic quality. Apart from the features and ergonomics of the modeler, a justified choice must also include considerations of sustainability and whether the import/export formats depend only on that specific modeler.
- Since criteria are not all equal and there are no perfect mechanisms, a feasibility analysis before any decision making is necessary. One possible approach in this case is to prioritize the criteria and adopt mechanisms based on that. Fortunately, not all mechanisms are needed simultaneously in the production process.

There are a few research directions that emanate from this work. The external and internal quality attributes in the proposed framework give us necessary but not sufficient conditions for pragmatic quality of UML artifacts; it is an open question as to *how far* are the necessary conditions from sufficiency. An investigation into it may lead us to discover other attributes, which would strengthen the framework further. According to the ISO/IEC 9126-1 Standard, other quality factors, namely usability and learnability, are related to this work. We anticipate a study of other cognitive dimensions and using that to build a general quality framework for UML artifacts as directions for future research. Addressing quality of artifacts in other special-purpose visual languages such as Feature Modeling for domain analysis, Object Role Modeling (ORM) for conceptual data modeling and Use Case Maps (UCM) for reactive systems would be of interest.

4. ACKNOWLEDGMENTS

The author would like to thank Terrill Fancott and Olga Ormandjieva (Concordia University, Canada) for their comments and feedback.

5. REFERENCES

[1] Ambler, S.W. *The Elements of UML Style*. Cambridge University Press (2003).

[2] Booch, G., Jacobson, I., and Rumbaugh, J. *The Unified Modeling Language Reference Manual, Second Edition.* Addison-Wesley (2005).

[3] Coad, P., Lefebvre, E., and Deluca, J. *Java Modeling in Color with UML: Enterprise Components and Process.* Prentice Hall (1999).

[4] Conradi, R., Mohagheghi, P., Arif, T., Hegde, L.C., Bunde, G.A., and Pedersen, A. Inspection of UML Diagrams using OORT - An Industrial Experiment. *European Conference for Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany, July 21-25, 2003*.

[5] Di Battista, G., Eades, P., Tamassia, R., and Tollis, I.G. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall (1999).

[6] Evitts, P. *A UML Pattern Language*. Macmillan (2000).

[7] Fenton, N.E. and Pfleeger, S.L. *Software Metrics: A Rigorous & Practical Approach.* International Thomson Computer Press (1997).

[8] Kamthan, P. A Framework for Addressing the Quality of UML Artifacts. *Studies in Communication Sciences*, 4, 2 (2004), 85-114.

[9] Keller, D. A Guide to Natural Naming. *ACM SIGPLAN Notices*, 25, 5, ACM Press (1990), 95-102.

[10] Kim, H. and Boldyreff, C. Developing Software Metrics Applicable to UML Models. *Sixth ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002), Malaga, Spain, June 11, 2002*.

[11] Lindland, O.I., Sindre, G., and Sølvberg, A. Understanding Quality in Conceptual Modeling. *IEEE Software*, 11, 2, IEEE Press (1994), 42-49.

[12] Nöth, W. *Handbook of Semiotics*. Indiana University Press (1990).

[13] Petre, M. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 38, 6, ACM Press (1995), 33-44.

[14] Schneidewind, N. F. and Fenton, N.E. Do Standards Improve Product Quality? *IEEE Software*, 13, 1, IEEE Press (1996), 22-24.

[15] Sunye, G., Pollet, D., Le Traon, Y., and Jezequel, J.-M. Refactoring UML Models. *Fourth International Conference on the Unified Modeling Language (<<UML>> 2001), Toronto, Canada, October 1-5, 2001*.

[16] Trong, T.T.D. A Systematic Procedure for Testing UML Designs. *Fourteenth International Symposium on Software Reliability Engineering (ISSRE 2003), Denver, USA, November 17-21, 2003*.

[17] Van Solingen, R. and Berghout, E. *The Goal/Question/Metric Method: A Practical Method for Quality Improvement of Software Development*. McGraw-Hill (1999).

[18] Unhelkar, B. *Verification and Validation for Quality of UML 2.0 Models*. John Wiley and Sons (2005).

Commonalities and Differences in Agile Development and User-Centered Design Methodologies

Muhammad Faraz Anwar
Department of Computer Science and Software Engineering
Concordia University, Montreal,
Quebec, Canada H3G 1M8
1-(514)-803-7271
mf_anwar@cse.concordia.ca

Ahmed Seffah
Department of Computer Science and Software Engineering
Concordia University, Montreal,
Quebec, Canada H3G 1M8
1-(514)-848-2424 ext. 3024
seffah@cse.concordia.ca

ABSTRACT
There are numerous methodologies and processes that govern the software development. Every process has its own features and some overlapping characteristics with other processes. In software development, there are two prominent philosophies that consider human/user involvement: *User-Centered Design* and *Agile Software development*. Both of these philosophies give a particular way of thinking about software engineering. Although these are two different philosophies, but we can draw some parallels between them while highlighting their differences. This study will enable researchers to find a common ground where the best of these methodologies could be put into practice.

Categories and Subject Descriptors
D.2.1 [Requirements/Specifications]; D2.2 [Design Tools and Tachniques].

General Terms
Design, Documentation, Human Factors, Theory.

Keywords
Requirements Engineering, User-Centered Design, Agile Methodologies, Commonalities, Differences, Application.

1. INTRODUCTION
In the development of small or large software projects, it is often desirable to adopt a methodology that is best suited for that particular project and organizational structure. The two major methodologies of Agile and UCD gives varied notions about how to approach problems. In this paper, we will take a look at the Agile software development methodologies and User-Centered Design methodology. Later we discuss the common and different points of these methodologies. Finally we present some scenarios where these methodologies could be applied singularly or mixed.

2. AGILE SOFTWARE DEVELOPMENT
The Software Development Process has undergone numerous revolutions since its inception. One of these revolutions is emergence of the philosophy of Agile Software Development. This philosophy is based on the notion that software development teams are focusing more on creating useless documentation and on the process itself rather than focusing on the product. The result is more delayed or failed projects. Agile philosophy and its supporting methodologies make sure that the development process is free from less fruitful rituals found in earlier processes. It gives new ideas for improving the communication between the team members and avoiding loopholes in development.

Agile philosophy took its current form with the emergence of agile manifesto in 2001 [2]. A group of practitioners came together to discuss new processes that were not heavyweight or documentation-oriented. What they came up with was a set of following values:

* Individuals and interactions over processes and tools

* Working software over comprehensive documentation

* Customer collaboration over contract negotiation

* Responding to change over following a plan

This philosophy and related methodologies have created lots of interest in professionals and academia. Abrahamson and others [1] have discussed major agile methods with respect to *Process, Roles and Responsibilities*, *Practices*, *Adoption and Experiences*, *Scope of use*, and finally *Current Research*. As a result of this approach, they have presented a definition and classification of agile methods, and different methods are compared with each other with respect to these aspects.

The most important work that was needed to be done, and was attempted by many researchers, is the adoption of agile values in conventional software engineering practices. Kutschera and Schafer [8] have presented a way to adopt agile methods in dynamic environments. Paetsch and others [11] have analyzed the role of agile methods in requirements engineering.

2.1 Highlights of Agile Development Philosophy
By definition, agile means: *marked by ready ability to move with quick easy grace* and/or *having a quick resourceful and adaptable character* [10]. The agile software development philosophy perfectly agrees with this definition. The philosophy advocates that the development process must always be ready to welcome change and yet must move with quick pace. The fruit of this thinking is more satisfied customers, developers with friendly rapport, and above all, good working software.

Most of the literature about this philosophy is produced by practitioners and consultants. As a result, this literature focuses on methodologies. Methodologies impose a disciplined process over software development with the aim of making development predictable and efficient [7]. However, the *Agile Manifesto* [2] gives a solid philosophical ground for methodologies. According to the manifesto, Agile Software Development is based on four values:

1) Individuals and interactions over processes and tools

Agile philosophy has a people-first orientation for software development [1]. That is, people are more important than processes. The software development process must suit the individuals who are developing the software. Some processes are better adapted by a group of developers in one culture but does not so in another culture or environment. According to Cockburn [4], people should not be treated as *components* that program. Rather, people are thinking and communicating beings suited for face-to-face communication. Therefore, one important breakthrough in agile methodologies is the importance of working with programmers' instincts though verbal communication (two-person teams in Extreme Programming, scrums in Scrum etc).

2) Working software over comprehensive documentation

Customers are always concerned with working software and have little interest in long documentations. Therefore, agile philosophy emphasizes on short but quick deliveries of working software. This does not mean that it discourages any kind of documentation, rather, the documentation should be done but only late in the process and when needed. The lack of documentation is the indication of two built-in characteristics of agile methods: (1) Agile methods are adaptive rather than predictive; i.e. they welcome change and also can change themselves according to the situation. (2) Agile methods are people-oriented rather than process-oriented; role of process is to support people (teams) in work [7].

3) Customer collaboration over contract negotiation

Although contracts are important from business point-of-view, they should not become a barrier against the communication between two parties. Agile philosophy ensures that the development team and client should collaborate with each other, especially over the requirements and do not freeze the requirements in the beginning of the project (this is particularly good for clients with changing requirements).

4) Responding to change over following a plan

Requirements change during the course of project. This fact has been taken graciously by agile philosophy and provided this important value in its manifesto. One way to control unpredictability due to changing requirements is 'iterations'. The length of iteration matters and dictates how often this change will be accommodated into design. XP and Scrum, including other methods, advise about the iteration length [7].

2.2 Focus on an Agile Method

In this section, we will focus on one major agile method that is used in industry and has been commonly studied. This will exemplify the highlights of the agile philosophy put forth in previous subsection.

2.2.1 Extreme Programming (XP)

Extreme Programming is the most popular agile methodology. It is based on four values namely: *communication, simplicity, feedback and courage*. Based on these values, about a dozen practices are suggested. These practices are not new; they are tested, tried but forgotten. XP offers a lifecycle process with phases: Exploration, Planning, Iterations to release, Production and finally death phase (when there is nothing more to implement). It is aimed for small and medium sized teams. Stress is put on team work and empowering developer to make decisions. [1, 7, 12].

Following table (table 1) summarizes the key concepts in XP and names and descriptions of major practices [1].

Table 1: Extreme Programming – Major concepts and practices

Key Concepts Respond to changing customer requirements Groupware-style development Communication, simplicity, feedback and courage	
Major Practices	Description
Planning game	Programmer decides effort, customer decides time for releases
Small releases	At least once every 2 to 3 months
Metaphor	A shared story guiding the development
Simple design	Design is simplest possible for implementation
Refactoring	Code is reviewed removed to discrepancies
Pair Programming	Programmers are always paired in a team of two.
Collective ownership	Anyone can change any part of code anytime
Continuous integration	A new piece of code is integrated into existing code as soon as it is ready.
40-hour week	Programmers work for no more than 40-hours per week.
On-site customer	A representative of customer is always present on programming site.
Coding standards	Coding rules and conventions exist and must be followed by all programmers.

3. USER-CENTERED DESIGN PHILOSOPHY

In any process of Software Engineering, design is an important phase. In this phase we consider the possible solutions of the problem, which was analyzed in analysis phase, and how to derive those solutions. During the design of the software, if we consider user as the focus of every activity, the end product will be liked by users. User-Centered Design approach advocates the same idea that since users are the ultimate goals of software, their role should be incorporated into the design process right from the beginning to the end.

To support this idea of user-centered design and to give solid guidelines that can fulfill this purpose, different researchers have devised several methods, e.g. [5]. These methods are based on a few key-concepts and advice some practices that will help in achieving a user-centered design. These key concepts also give interesting insights into how ideas from other Software Engineering practices and other fields like psychology could be adopted for a User-Centered Design.

3.1 Highlights of the User-Centered Design Philosophy

The philosophy of User-Centered Design and HCI has roots in disciplines of psychology, sociology, industrial design, graphic design and others. This amalgamation of paradigms has made UCD an interesting field. In software engineering, this is taken on purely engineering approach and several methods are derived from this philosophy that makes the software development closer to user needs.

According to Ferre [6], the iterative approach in UCD philosophy is crucial. It is impossible to make a correct design in first attempt due to the complexity of human behavior. Iterations, therefore, play a key role in defining user needs and refining them to render them useful.

The most obvious highlight of this philosophy is active user involvement. Unless the user is involved from the start of the software development process, it is difficult to make a system that is completely user-satisfying. The UCD philosophy enjoins the development team to contact user on each and every step of the process, get their feedback, inform them of the status of the progress, and above all, evaluate short deliveries with them.

One important concept in UCD philosophy is proper understanding of user and tasks [6]. Understating users is quite obvious, but for tasks, the UCD philosophy says that these are also important to understand. The viewpoint to look at tasks, in case of UCD, is different. In conventional methods, tasks are looked upon as features to implement. In UCD, tasks are set of actions that a user has to perform to achieve a goal. The viewpoint, thus, has shifted from system/software to user/human. This important shift in paradigm has enabled developers and designers to put themselves in user's shoes and see what user will have to do for hitting that goal. They thus design systems that are close to user's expectations from the system.

Users are humans. Humans are affected by their environment, so do users. The UCD philosophy also emphasizes the need to study user's environment and take decisions accordingly. Context is defined as the surroundings of users while they are using the system. Contextual inquiry is thus deemed important in UCD philosophy. Users' detailed sketch includes their education, exposure to similar systems, social status etc. make another important factor in their behavior with the system. UCD thus underlines the importance of understating the users themselves.

3.2 Focus on a Major UCD Method

User-Centered Design is a topic of research of a great many software engineering scholars and practitioners. Here in this section, we will consider Scenario-based design to give an idea of how their key-concepts and major practices can make the process and thus the product closer to the needs of end-users.

3.2.1 Scenario-Based Design

Scenarios are short stories telling about the use of system by the users. The Scenario-based design puts scenarios in focus and derives solution based on requirements gathered from them. In scenario-based design, descriptions of how people accomplish tasks are a primary working design representation [3]. Collecting scenarios involves users in telling stories about their use of the system. To collect and elicit scenarios, pictures, videos, and storyboards are used. Table 2 summarizes key concepts and major practice in scenario-based design.

Table 2: Scenario-based design – Major concepts and practices

Key Concepts Scenario Video, pictures, storyboards	
Major Practices	Description
User Involvement	Involve users to make and refine scenarios

4. DISCUSSION: COMMONALITIES AND DIFFERENCES

As we have seen in preceding section, UCD and agile software development are two different philosophies, developed by different people at different times. Yet, they have many aspects in common. In this section, we will shed some light on the commonalities and differences between these two philosophies. The summary of the following discussion is presented in Table 3.

Two values of agile manifesto are: (1) Individuals and Interactions over processes & tools. (2) Customer collaboration over contract negotiation. These values are in harmony with the UCD concepts of putting the emphasis on individuals (users and stakeholders). Stakeholders are people who have any interest in the software. The end-users are one of these stakeholders. In agile, any stakeholder (called Customers) is given same importance and is encouraged to interact with the development team.

On the other hand in UCD, the end-user is the primary concern of the usability team since it is the end-user who is going to interact with the user-interface of the software. In the context of agile methodologies, Individuals also refer to development team members with different skill-sets.

During discussion about the role of overlapping lifecycle phases, Mayhew [9] points out that optimal implementation of the lifecycle requires full participation of all teams. In traditional software engineering, however, people of different skill-sets work on their own part of lifecycle and communication is done through documents. Instead, if all people work together in each phase of the project, they can input their expert advice and raise their concern at the right time. This idea of collaboration in Usability lifecycle resonates perfectly with these agile values.

The other two values of agile philosophy are: (1) Working software over comprehensive documentation, and (2) Responding to change over following a plan. These values are not very common in UCD circles. In UCD, emphasis is put on getting the user-goals and requirements in written form. Style-guides are suggested to be made/updated after every major phase [10].

Prototypes are encouraged to be made and evaluated long before the actual product is produced.

In agile methodologies, a working, deliverable version of software is always desirable and documentation is delayed to be done as late as possible. Change tolerance is also projected in UCD, but responding very quickly to change sacrificing the process is not advocated. Rather, this change management is incorporated into the UCD process in the form of short, frequent iterations and user evaluations.

Requirement fixing is discouraged in both philosophies. Customers (users in UCD terms) are encouraged to collaborate with the development team. During this collaboration, users sometimes realize that what they termed as necessary in the system are not too necessary and vice versa. At this point, the development team adjusts the requirements and other plans to accommodate these changes. Change in environment can also sometimes make change necessary.

Both philosophies stress customer satisfaction and have a people-first orientation for software development. This causes their corresponding methods to have tendency to come together and provide efficient methods for software development.

Another aspect that is common in both philosophies is the iterative approach of lifecycle. Due to the complexity of human behavior, it is impossible in UCD to create a design that is correct in the first attempt. In agile, similarly, iterations are a way to manage changes and refining the product.

We can summarize the above discussion in a table (table 2.2). It juxtaposes the two philosophies in terms how one aspect in UCD is considered in agile philosophy.

Table 3: Summary of Commonalities and Differences in agile and UCD philosophies

Agile Software Development	User-Centered Design
Customer collaboration	User involvement
Stakeholder satisfaction	End-user satisfaction
Developer as focus in process	End-users are focus in process, not developers
Documentation as late as possible	Documentation after every major phase
Quick delivery of working software	Frequent evaluation of prototypes
Process should be flexible enough to accommodate different projects	Process should be tailored for different organizations
Change should be reflected in next delivery	Change should be accommodated in next design iteration
Choice of which task to perform first	Choice of which technique to perform tasks

5. WHEN TO USE WHAT? SUGGESTIONS FOR CHOOSING THE RIGHT METHOD

It is often confusing for developers to decide when to use what method. Whether the agile methods are good for their project? How necessary and feasible it is to spend time in UCD practices? Will a quick heuristic evaluation be enough for the product? In the light of the above discussion, we will try to present some scenarios where different methods would be helpful. However, it is the judgment applied to the project on the development time which will be most correct for the project.

Scenario 1: Agile methods

If the project and company has following statements true:

* It is a small- to medium-sized project.

* The time limit is short.

* Customer wants quick deliveries.

* Requirements seem to change frequently.

* Team is not big i.e. 3-7 people working on the project.

It is better in these situations to use agile development methods like Extreme programming which work very well for small- to medium-size projects with tight deadlines and changing requirements.

Scenario 2: UCD methods

If the project has following characteristics:

* Time limit is not too tight.

* Product is to be used by people of varied backgrounds.

* There are some special deployment needs of the system, like embedded systems or kiosks.

* Product is to be used by people with some special needs.

* Ease-of-use is emphasized by customer.

In such cases, it is better to start off the project with proper user-centered design process and use usability analysis and design methods. Testing should be done after every development iteration and prototyping should be used to verify the design of the system with users.

Scenario 3: Mix of Agile and UCD

Agile and UCD methods could be used together. If the project has following characteristics, consider mixing both practices.

* Customer is available for frequent interaction during development.

* Project deadline is not too tight.

* Team is small to medium sized.

* Usability matters for the users of the system.

In such cases, its feasible to have small agile teams developing small iterations and one team ensures usability by helping in designing a good user-interface. As soon as iteration is ready for delivery, a quick usability test would be worthwhile.

5. CONCLUSIONS

The two prominent philosophies in software engineering that emphasize user involvement during development are Agile and User-Centered Design. Four values that the agile philosophy is based on are: *Individuals and interactions over processes and tools, Working software over comprehensive documentation, Customer collaboration over contract negotiation, Responding to change over following a plan.* Agile methods are mainly devoted towards the implementation phase of software development lifecycle. User-centered design, on the other hand, involves users/human right from the beginning of software development lifecycle. Its methods include interaction with users frequently to get their requirements. There are several differences and commonalities in these two philosophies. The commonalities indicate that the methods of corresponding philosophies could be acted upon on a common ground. The differences highlight the areas of further research. In the end, we present some suggestions on when to use agile and when to use UCD and when both together.

6. REFERENCES

[1] Abrahamson, P., Salo, O., Ronkainnen, J. and Warsta, J. (2002) Agile software development methods - Review and analysis, ESPOO 2002, VTT Publications, pp.107.

[2] Beck, K., Beedle, M., Bennekum, A. V., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D. (2001), *Agile Manifesto*, accessed: March 2004, available at: http://www.agilemanifesto.org/

[3] Carroll, J. M. (2000) *Making Use: Scenario-based design of Human-Computer Interactions* In Proceedings of the Designing interactive systems: processes, practices, methods, and techniques, December 2000, New York, NY, USA, ACM Press New York, USA, pp. 4.

[4] Cockburn, A. (2000) *Characterizing People as Non-Linear, First-Order Components in Software Development* In Proceedings of the 4th International Multi-Conference on Systems, Cybernetics and Informatics, June 2000, Orlando, Florida, pp. 19.

[5] Constantine, L. L. and Lockwood, L., A.D. (2002) 'Usage-Centered Engineering for Web Applications', *IEEE Software*, vol. 19, issue. 2, pp. 42 - 50. [6] Ferre, X. (2003) *Integration of Usability Techniques into the Software Development Process* In Proceedings of the International Conference on Software Engineering - ICSE 2003, May 3-10, 2003, Portland, Oregon, USA, pp. 28-35.

[6] Ferre, X. (2003) *Integration of Usability Techniques into the Software Development Process* In Proceedings of the International Conference on Software Engineering - ICSE 2003, May 3-10, 2003, Portland, Oregon, USA, pp. 28-35.

[7] Fowler, M. (2000), *The New Methodology*, accessed: March 2004, available at: www.martinfowler.com/articles/newMethodology.html, April 2003

[8] Kutschera, P. and Schafer, S. (2002), *Applying Agile methods in rapidly changing environments*, accessed: 2004, available at: http://jeckstein.com/papers/Agile%20Methods%20-%20Steffen%20Schaefer%20&%20Peter%20Kutschera.pdf

[9] Mayhew, D. J. (1999) The Usability Engineering Lifecycle: A practitioner's handbook for User Interface Design, Morgan Kaufmann Publishers, Inc., San Francisco, California.

[10] Merriam-Webster (1982), *Merriam-Webster Online Dictionary*, accessed: 2004, available at: www.m-w.com July-23-2002

[11] Paetsch, F., Eberlein, D. A. and Maurer, D. F. (2003) *Requirements Engineering and Agile Software Development* In Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03), 9-11 June 2003, IEEE Computer Society, pp. 308 - 313.

[12] Wells, D. (2004), *Extreme Programming: A gentle introduction*, accessed: March 2004, available at: http://www.extremeprogramming.org/, February 28, 2004

Engineering the Requirements in User-Centered Design and Agile Development Methodologies

Muhammad Faraz Anwar
Department of Computer Science and
Software Engineering
Concordia University, Montreal,
Quebec, Canada H3G 1M8
1-(514)-803-7271
mf_anwar@cse.concordia.ca

1. ABSTRACT

Among all phases of software development, requirements engineering has its unique place. It is the phase of requirements engineering where the development team has a chance to understand the domain of the system. There are many methodologies known in the software engineering communities which place different degrees of emphasis on requirements engineering process. In this paper, we will provide an analysis of how requirements engineering is treated in two major software engineering paradigms of User-Centered-Design and Agile. We do this by providing a critique of the most recent research pertaining to these two paradigms. Consequently, we present a framework of requirements engineering which is founded in user-centered design principles. The insights of this paper will help development teams to choose a suitable method for the requirement engineering phase for their projects.

2. Categories and Subject Descriptors

D.2.1 [Requirements/Specifications]; D2.2 [Design Tools and Tachniques].

3. General Terms

Documentation, Human Factotrs, Theory.

4. Keywords

Requirements Engineering, User-Centered Design, Agile Methodologies.

1. INTRODUCTION

User-centered requirements engineering is a significant field in software engineering. The purpose of every software is to solve certain problems or to provide an easy means to perform different tasks. Knowing these problems or tasks, thus, is the most logical first step in development of software systems. The field of requirements engineering deals with this first step in software development. It is the topic of study for many researchers e.g. [8, 17, 18, 19].

One common aspect in almost all interactive software systems is that they should support human experiences while providing task- and context-aware interaction. Designing such interactive systems is no trivial task; therefore, trial-and-error is not sufficient and there has to be a well-defined method for collecting and analyzing requirements. Different methods are needed for different activities in requirements engineering. These activities could be grouped as elicitation, specification and validation. The methods for requirements engineering, besides being well-defined, must also be usable by software engineers. Different software engineering methodologies have proposed different methods for requirements engineering and put varying degrees of emphasis on this important phase of software development.

Requirements are collected from the domain the software is to be used in. Besides all the current methods that exist, one thing that should be of principal concern in requirements engineering methods is the focus on end-users. Orr argues that job of requirements engineer is to discover what users really need [18]. Since the product is made for human users, its success depends upon how well humans can use the system. The focus of requirements engineering and design in User-Centered Design and User-Centered Requirements Engineering is user and not the functionality [16].

There are two levels of human interaction involved in user-centered requirements engineering methods. First, the requirements engineers use these methods to interact with end-users to collect domain knowledge, and secondly, since the methods are used by human software engineers, the process itself must also be easy for humans to be followed. This is where agile software methodologies come into play. Agile development is a well-established idea practiced by many people in software engineering community (e.g. [7, 12, 15]). Its methodologies, for example Extreme Programming, advocate the involvement of users during the software development and flexibility in the process to adapt with human work psychology. Agile aspects in user-centered requirements engineering process, we believe, can give a solid and comfortable ground for user-centered requirements engineers.

1.1 An Overview of User-Centered Design Methodologies

The philosophy of User-Centered Design and HCI has roots in disciplines of psychology, sociology, industrial design, graphic design and others. This amalgamation of paradigms has made UCD an interesting field. In software engineering, this is taken on purely engineering approach and several methods are derived from this philosophy that makes the software development closer

to user needs. Here it would be worthwhile to mention that although these methodologies are called user-centered *design*, they are not limited to the design phase of development. As we will see later in this paper, UCD methods put rightful emphasis on requirements engineering as well.

The most obvious highlight of these methodologies is active user involvement. Unless the user is involved from the start of the software development process, it is difficult to make a system that is completely user-satisfying. The UCD philosophy enjoins the development team to contact user on each and every step of the process, get their feedback, inform them of the status of the progress, and above all, evaluate short deliveries with them.

One important concept in UCD philosophy is proper understanding of user and tasks [13]. Understating users is quite obvious, but for tasks, the UCD philosophy says that these are also important to understand. The viewpoint to look at tasks, in case of UCD, is different. In conventional methods, tasks are looked upon as *features to implement.* In UCD, tasks are set of actions that a user has to perform to achieve a goal. The viewpoint, thus, has shifted from system/software to user/human. This important shift in paradigm has enabled developers and designers to put themselves in user's shoes and see what user will have to do for hitting that goal. They thus design systems that are close to user's expectations from the system.

Users are humans. Humans are affected by their environment, so do users. UCD methodologies also emphasize the need to study user's environment and take decisions accordingly. Context is defined as the surroundings of users while they are using the system. Contextual inquiry is thus deemed important in UCD philosophy. Users' detailed sketch includes their education, exposure to similar systems, social status etc. make another important factor in their behavior with the system. UCD thus underlines the importance of understating the users themselves.

Following are major methodologies of UCD:

* Scenario-based design

* Contextual design

* Usage-centered design

The highlights of UCD methods could be summarized in the following table.

Table1: Highlights of UCD methods

Characteristics / Highlights	Present in Agile?
Roots in disciplines of psychology, sociology and industrial design.	No
User involvement in every phase of software development.	Yes
Encompasses requirements engineering activities.	Yes
Shifts paradigm from functions to implement to tasks to support.	No

1.2 An Overview of Agile Methodologies

Agile methodologies are based on the notion that software development teams are focusing more on creating useless documentation and on the process itself rather than focusing on the product. The result is more delayed or failed projects. Agile philosophy and its supporting methodologies make sure that the development process is free from less fruitful rituals found in earlier processes. It gives new ideas for improving the communication between the team members and avoiding loopholes in development.

Agile philosophy has matured over several years. The emergence of agile manifesto in 2001 [7] was a major milestone in the history of agile methodologies. A group of software engineers and practitioners of the field discussed the best practices that they had developed through time. These practices and methodologies were not *heavyweight* or documentation-oriented. Together they decided on a set of following values:

*** Individuals and interactions** over processes and tools
*** Working software** over comprehensive documentation
*** Customer collaboration** over contract negotiation
*** Responding to change** over following a plan

This philosophy and related methodologies have created lots of interest in professionals and academia. Abrahamson and others [1] have discussed major agile methods with respect to Process, Roles and Responsibilities, Practices, Adoption and Experiences, Scope of use, and finally Current Research. Paetsch and others [19] have analyzed the role of agile methods in requirements engineering.

Some most common agile methodologies are:

* Extreme Programming
* Scrum
* Crystal Methods
* Dynamic System Development Method (DSDM)

Highlights of agile methodologies could be summarized as below.

Table 2: Highlights of Agile methodologies

Characteristics / Highlights	**Preesent in UCD?**
Places importance on individuals (users and developers).	Yes
Documentation is kept to minimum, deliveries are kept at maximum.	No
Customer is always accessible.	Yes

2. REQUIREMENTS ENGINEERING IN USER-CENTERED DESIGN

Requirements engineering is an important phase in User-Centered Design. Unless it is clear what users want, it is impossible to make a system that can satisfy users. Requirement engineering in UCD

is often divided in several phases. Each of these phases plays a role in building up requirements which are vague in the beginning. These phases are usually characterized as Elicitation, Analysis, Design and Validation. According to Cox [11], common activities of requirement engineering process are:

* Project Inception
* Requirement Elicitation
* Requirement Analysis
* Requirement Discovery
* Specification
* Interface Design
* Validation

Project Inception and Requirement Elicitation can be grouped in *Elicitation* phase; activities of Requirement Analysis and Discovery can be grouped into *Analysis* phase; activities of Specification and Interface Design can be grouped into *Design* phase; and Validation is itself a phase. Requirement Discovery is inventing new requirements from existing ones. We can validate the user requirements using the prototypes.

An important facet of requirement analysis and elicitation is context analysis. Context analysis is going to field with users and see how they use the current system. This practice gives useful insights into future system's functional and non-functional requirements. In automotive industry, for example, developing functionality from scratch is a rare practice [25]. Studying already present systems and analyzing context are crucial steps in requirement engineering. The study of user context in requirements engineering is also highlighted in process diagram by UPA [24].

Jokela [12] identifies the context in which system is to be used in terms of:

* Characteristics of intended users
* Tasks users need to perform
* Environment in which the users are to use the system.

This information collected in context analysis provides essential insights on users and their requirements.

International Standards Organization (ISO) established the ISO 13407 standard for User-Centered Design process in 1999. This document is based on the definition of usability in ISO 9241-11 and tries to formulate a process that can fit into conventional software engineering processes as well.

Jokela and others [12] discusses the ISO 13407 in detail. According to them, ISO 13407 shows limited guidance for *designing* usability. What it emphasizes is guidance for user and environment/context of use. It also has limited guidance for user goals and measures and the focus is on theoretical aspects of usability, rather than detailed coverage of methods and techniques.

ISO 13407 describes UCD from four different aspects, which are: Rationale, Principles, Planning and Activities of UCD. In *Rationale*, it explains the benefits of UCD such as reduced cost, increased satisfaction and productivity of users. *Principles* that usability is based on are active user involvement, appropriate allocation of functions between user and technology and multi-disciplinary design. *Planning* tries to fit the usability with the conventional software engineering process [12]. Another aspect of usability according to ISO 13407 is the *activities* of UCD. These activities include:

* Understanding the context of use
* Specifying user and organizational environment
* Producing design solution
* Evaluating design against requirements

Of these activities, the first two: *Understanding context* and *specifying user* are especially relevant to requirement engineering process.

3. REQUIREMENTS ENGINEERING IN AGILE METHODOLOGIES

The heart of agile methodologies lies in changing requirements. The agile philosophy advocates that the requirements should never be frozen; instead, it always welcomes change and adjusts the software according to new requirements. There are different approaches to address the requirement management in different agile methods.

The traditional requirement engineering approaches and agile methods agree on the importance of stakeholder involvement. The requirements are discussed in face-to-face meetings with customers rather than through formal documents; the reason is that agile philosophy is more people-oriented than process-oriented. The customers (or customer representatives) are encouraged to be present on the development site during all phases of development. This customer is often assumed to have all the knowledge and authority in the project, which is rarely the case [19].

The common requirements engineering phases of elicitation, analysis and validation are present in all agile processes but with different names and do not have crisp boundaries. Techniques used are also different.

In agile methodologies, creating complete and consistent requirements documents is not considered feasible or cost effective [19]. This makes agile methods more adaptive to change rather than being predictive of user requirements. This is considered a good quality in agile terms but certain traditional approaches discourage this idea because it makes the software development process very unpredictable.

In Extreme programming, customer reviews all the requirements and sets priority for implementation. It enables software to be developed without disruption despite of vague or constantly changing requirements. There is no artifact, however, to store requirements besides user stories. User stories are similar as scenarios and help record useful non-functional as well as functional requirements.

In Scrum, the requirements that are currently known are saved in Product Backlog list. This is a way to store, but not a tool to collect requirements. The Sprints in the Scrum method involves requirement phase along with other phases (there are several sprints in a Scrum method lifecycle).

In Crystal Orange, one of the methods in Crystal methods set, a requirements document is required; requirements to be implemented are decided before every increment starts. Feature Driven Design does not explicitly address the issue of gathering and managing requirements [1].

4. FRAMEWORK SUPPORT FOR REQUIREMENTS ENGINEERING

Having seen two major methodologies of software engineering and how they treat the requirement engineering, we will now look at a framework that is proposed under the umbrella of user-centered design. The framework help the practitioners apply the ideas and methods of UCD with the help of scenarios. A detailed discussion of this framework is found in [3].

4.1 An Overview Of Sucre Framework

The *Scenario-based User-Centered Requirements Engineering* (SUCRE) is a requirement engineering framework based on Scenarios. This framework was developed by Alsumait [3] in department of Computer Science, Concordia University. According to this framework, the requirements are captured and recorded in scenarios which are represented with use case-maps. The use-case maps are semi-formal notations to represent scenarios and could be used to elilcit requirements. Figure 1 shows the structure of SUCRE framework.

This framework is evolved from ACUDUC which in turn is derived from RESPECT framework (Figure 2). A brief introduction of ACUDUC and RESPECT is given in following paragraphs.

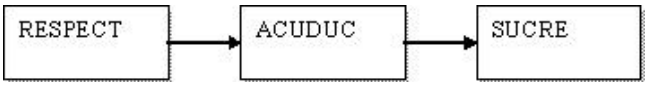

Figure 2: Evolution of SUCRE framework

The RESPECT (REquirements SPECification in Telematics) [17] gives a framework for requirements engineering. The requirements are achieved with this framework in four phases: *Phase I – User context and early design, Phase II – Prototype and user test, and Phase III – User requirements documentation*. This framework is exceptionally good in proposing templates and forms that could be used in contextual analysis.

The ACUDUC (Approach Centered on Usability and Driven by Use Cases) framework which combines use-cases with RESPECT, is proposed by Seffah and his team [21]. It discusses the following key activities in requirements engineering:

* Summarizing the system
* Gathering context of use.
* Functional requirement, including UI widgets
* Reviewing and Validating

These activities are defined and validated through industrial projects. Anwar [5] has presented a roadmap for user-centered

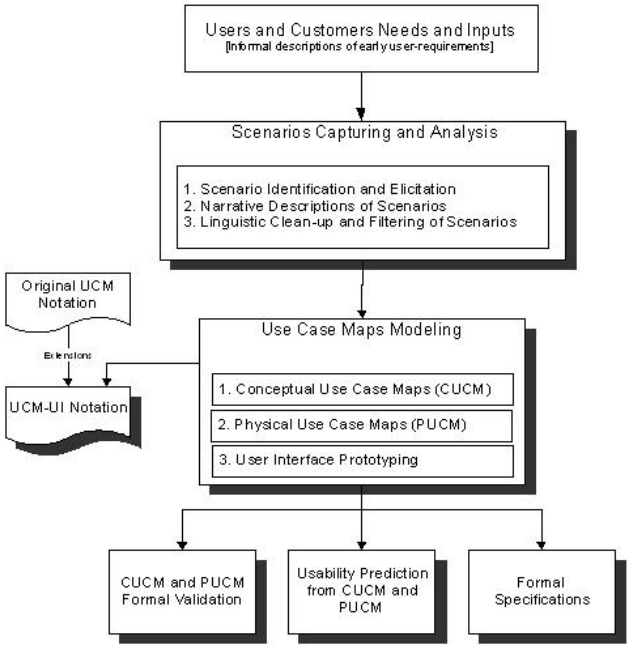

Figure 1: Structure of SUCRE framework [3]

Requirements engineering which includes these steps as its foundation.

An important work for Usability Requirements is done by Seffah and Alsumait [4]. They have showed that Use-Case Maps (UCMs) work well for user-interface requirement engineering by extending the basic notation of UCMs and fragmenting the UCM design process into two steps, namely, the Conceptual Use-Case maps and Physical Use-Case maps. This extension of UCMs: CUCMs together with PUCMs, make up the SUCRE framework [3]. This framework presents an approach for UI Requirements Engineering through Scenarios and UCMs.

The role of scenarios in requirements engineering is also studied by several researchers [9, 10, 2, 20, 22, 6]. According to them, scenarios have the potential to play important role in requirements engineering. Some have proposed a scenario-based model e.g. Sutcliffe and Ryan [23]. Scenarios are beneficial in re-use of knowledge in requirements engineering because they store a wealth of domain knowledge in them that can be understood by people of every level of expertise in development team and stakeholders. Scenarios can also be used to derive mid-fidelity prototype, like storyboards, that are beneficial in requirements analysis. A process for this type of derivation is proposed by Anwar [5].

5. CONCLUSION

Requirements engineering is treated differently by different methodologies, although it is a very important phase during software development. User-Centered Design puts more emphasis on requirements engineering than agile methodologies do. Agile philosophy believes in incorporating changing requirements during the implementation. There are several methods for User-centered requirements engineering, SUCRE being one of them. SUCRE is an evolution of ACUDUC framework which

incorporates use-cases into the RESPECT framework. SUCRE framework is based on scenarios and employs use-case maps to represent these scenarios. There is also a prototype-derivation process which compliments the SUCRE framework.

The discussion in this paper about the treatment of requirements in UCD and agile methodologies wil help practitioners as well as learners decide what to expect in terms of requirements from both of these methodologies. This is specially useful when, in the analysis phase, development team has to decide among the methods to use for requirements engineering. For this, we suggest that if the project has a lot of user interaction involved, such as a web application, it is better considering UCD methods instead of agile methods. In case of projects where functionality is important and quick deliveries are required, agile methodologies would serve better. Attempting to employ lengthy UCD process when application must be delivered to the customer quickly would not be a wise decision. However in some cases, these two methodologies can compliment each other, such as while using extreme programming and customer is available on-site. Besides the practice of these methods in industry, those who are learning could benefit from this discussion since they can know the important aspects of each of these methodologies in domain of requirement engineering.

6. REFERENCES

[1] Abrahamson, P., Salo, O., Ronkainnen, J. and Warsta, J. (2002) Agile software development methods - Review and analysis, ESPOO 2002, VTT Publications, pp.107.

[2] Achour, C. B. (1998) *Writing and correcting textual scenarios for system design* In Proceedings of the Ninth International Workshop on Database and Expert Systems Applications, 26-28 Aug. 1998, Vienna, Austria, pp. 166 - 170.

[3] Alsumait, A. (2004) *User Interface Requirements Engineering: A Scenario-Based Framework*, PhD. Thesis in Dept. of Computer Science and Software Engineering, Concordia University, Montreal.

[4] Alsumait, A., Seffah, A. and Radhakrishnan, T. (2003) *Use Case Maps: A Visual Notation for Scenario-Based User Requirements* In Proceedings of the 10th International Conference on Human - Computer Interaction, June 22 - 27, Crete, Greece.

[5] Anwar, M. F. (2005) *An Agilized Roadmap for User Centered Requirements Engineering and Prototype Generation*, Masters Thesis in Dept. of Computer Science and Software Engineering, Concordia University, Montreal.

[6] Bai, X., Tsai, W. T., Paul, R., Feng, K. and Yu, L. (2002) *Scenario-based modeling and its applications* In Proceedings of the Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems, 7-9 Jan. 2002, pp. 253 - 260.

[7] Beck, K., Beedle, M., Bennekum, A. V., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D. (2001), *Agile Manifesto*, accessed: March 2004, available at: http://www.agilemanifesto.org/

[8] Carroll, J. M., Rosson, M. B., Chin, G. J. and Koenemann, J. (1998) 'Requirements development in scenario-based design', *IEEE Transactions on Software Engineering*, vol. 24, issue. 12, pp. 1156 - 1170

[9] Carrol, J. M. (1999) *Five reasons for scenario-based design* In Proceedings of the 32nd Annual Hawaii International Conference System Sciences, 1999. HICSS-32, 5-8 Jan. 1999, pp. 11.

[10] Carroll, J. M. (2000) *Making Use: Scenario-based design of Human-Computer Interactions* In Proceedings of the Designing interactive systems: processes, practices, methods, and techniques, December 2000, New York, NY, USA, ACM Press New York, USA, pp. 4.

[11] Cox, K. (2000) *Fitting scenarios to the requirements process* In Proceedings of the 11th International Workshop on Database and Expert Systems Applications 2000 (DEXA'00), 4-8 Sept. 2000, Greenwich, London, U.K., IEEE Computer Society Press, 2000, pp. 995 - 999.

[12] Fowler, M. (2000), *The New Methodology*, accessed: March 2004, available at: www.martinfowler.com/articles/newMethodology.html, April 2003

[13] Ferre, X. (2003) *Integration of Usability Techniques into the Software Development Process* In Proceedings of the International Conference on Software Engineering - ICSE 2003, May 3-10, 2003, Portland, Oregon, USA, pp. 28-35.

[14] Jokela, T., Iivari, N., Matero, J. and Karukka, M. (2003) *The Standard of User Centered Design and the Standard definition of Usability: Analyzing ISO 13407 against ISO 9241-11* In Proceedings of the Latin American conference on Human-computer interaction, Rio de Janeiro, Brazil, ACM Press New York, NY, USA, pp. 53 - 60.

[15] Kutschera, P. and Schafer, S. (2002), *Applying Agile methods in rapidly changing environments*, accessed: 2004, available at: http://jeckstein.com/papers/Agile%20Methods%20-%20Steffen%20Schaefer%20&%20Peter%20Kutschera.pdf, July-23-2002

[16] Lauesen, S. (1997) *Adding Usability to Software Engineering* In Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction, INTERACT '97, 14th-18th July 1997, Sydney, Australia, Chapman & Hall 1997.

[17] Maguire, M. C. (1998), *RESPECT 5.3: User-Centred Requirements Handbook*, accessed: 31st March 2004, available at: http://www.ejeisa.com/nectar/respect/5.3/contents.htm, 29 June 1998

[18] Orr, K. (2004) 'Agile requirements: opportunity or oxymoron?' *IEEE Software*, vol. 21, issue. 3, pp. 71 - 73.

[19] Paetsch, F., Eberlein, D. A. and Maurer, D. F. (2003) *Requirements Engineering and Agile Software Development* In Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03), 9-11 June 2003, IEEE Computer Society, pp. 308 - 313.

[20] Pohl, K., Weidenhaupt, K., Jarke, M. and Haumer, P. (1998) 'Scenarios in system development: current practice', *IEEE Software*, vol. 15, issue. 2, pp. 34 - 45.

[21] Seffah, A., Djouab, R. and Antunes, H. (2001) *Comparing and reconciling Usability-Centered and Use Case-Driven Requirements Engineering Process* In Proceedings of the 2nd Australasian conference on User interface, Queensland, Australia, IEEE Computer Society Washington, DC, USA, pp. 132 - 139.

[22] Sutcliffe, A. G., Maiden, N. A. M., Minocha, S. and Manuel, D. (1998) 'Supporting scenario-based requirements engineering', *IEEE Transactions on Software Engineering*, vol. 24, issue. 12, pp. 1072 - 1088.

[23] Sutcliffe, A. G. and Ryan, M. (1998) *Experience with SCRAM, a Scenario Requirements Analysis Method* In Proceedings of the Proceedings of the Third International Conference on Requirements Engineering, 1998., 6-10 April 1998, Colorado Springs, pp. 164 - 171.

[24] UPA (2002), *Usability Professionals Association website*, accessed: March 2004, available at: http://www.usabilityprofessionals.org

[25] Weber, M. and Weisbrod, J. (2003) 'Requirements Engineering in Automotive Development: Experiences and Challenges', *IEEE Software*, vol. 20, issue. 1, pp. 16 - 24.

Tool Support for an Evolutionary Design Process using XML and User-Interface Patterns

Peter Forbrig, Andreas Wolff,
University of Rostock
Institute of Computer Science
Albert Einstein Str. 21,
18059 Rostock, Germany
[rusty|pforbrig]@informatik.uni-rostock.de

Anke Dittmar, Daniel Reichart
University of Rostock
Institute of Computer Science
Albert Einstein Str. 21,
18059 Rostock, Germany
[ad|dr007]@informatik.uni-rostock.de

ABSTRACT
Design patterns are very helpful to develop well-structured software. This fact is widely accepted by the software engineering community. The same kind of support is expected by UI-Patterns.

In this paper we discuss an approach integrating use of patterns into a model-based development process. It will be shown a GUI editor that can be extended by the feature of managing the application of pattern-instances.

As an example the pattern of a wizard is used to improve an existing user interface by manipulating different models (UI and dialog model).

Additionally, useful XML-languages are discussed. Especially tool support for developing parts user interfaces from class diagrams and their integration into dialog models is presented.

Author Keywords
Model-Based Design, Task Models, Object Models, Patterns, XUL, XIML.

ACM Classification Keywords
HCI

INTRODUCTION

Along with the enhancing capabilities of mobile devices model-based development of software systems becomes popular. In this domain varying platforms have to be supported in an economic way by new interactive applications.

It is especially necessary to design user interfaces in an abstract way because there is a diversity of different platforms with specific features. Models help to derive specifications of interactive systems, and in particular, of user interfaces. We consider model based software development as a sequence of transformations of models that is not performed in a fully automated way, but supported by humans using interactive tools.

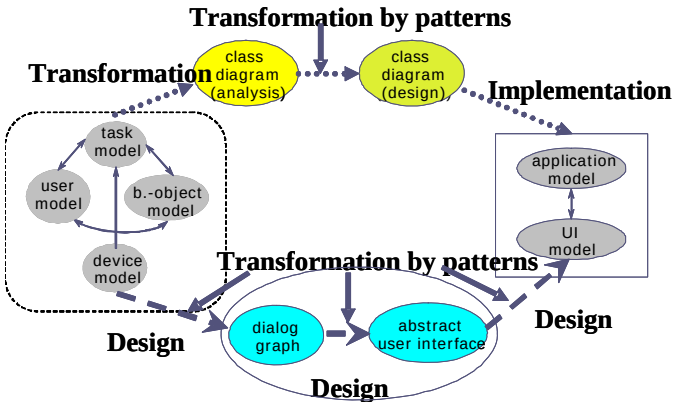

Figure 1 - General view on a transformational model-based development process.

We also think these persons, i.e. software engineers and user interface designers; have to base their work on the same models. Our work is especially focused on methods and tools supporting transformations by patterns.

In this paper we focus on the user interface development process.

The paper is structured in such a way that after discussing some related work an introduction of our development process will be given. A specific user interface for a small application is developed. Following that, the example user interface will be further enhanced, by introducing our proposal of how to integrate the appliance of ui-patterns within this development approach exemplarily.

At the end we will give an outlook of future work to be done.

RELATED WORK

Our work is very much related to the "mapping problem" that was first mentioned by Puerta and Eisenstein [17]. They stated that the mapping problem is the key problem to make model-based development acceptable for programmers. The mappings mentioned include only mappings from abstract to concrete models and between models of the same level. No mappings from concrete to abstract models are mentioned in their paper. This was carefully analysed by Clerckx, Luyten and Coninx [2]. They have classified five mechanisms to solve the mapping problem.

1. Model derivation
2. Partial model derivation
3. Model linking
4. Model modification
5. Model update.

Limbourg and Vanderdocnckt [11] address the problem by supporting transformation of abstract models to more concrete ones by graph grammars. The user interface specification is based on UsiXML [24].

"UsiXML (which stands for USer Interface eXtensible Markup Language) is a XML-compliant mark-up language that describes the UI for multiple contexts of use such as Character User Interfaces (CUIs), Graphical User Interfaces (GUIs), Auditory User Interfaces, and Multi-Modal User Interfaces. In other words, interactive applications with different types of interaction techniques, modalities of use and computing platforms can be described in a way that preserves the design independently from peculiar characteristics of physical computing platform"[24]. It seems to be that UsiXML could be a living standard to express models. It can play the role, which XIML [29] originally wanted to gain.

The initiative for XIML started in 1999 and was focused on device-independence primarily of mobile devices. XIML is model-based but it needs a specific tool (converter) to create a specific type of user interface. Our tool DiaTask [6] was developed to make use of XIML. Task models, user models, and object models with our metaphor of artefacts and tools are represented as XIML specifications. However, there seems to be no further support for XIML. Still there is a lack of tool support. This is especially true for designing a concrete user interface. That was the reason for our group to look for user interface specifications, which are already supported by tools. We found XUL as a candidate for that.

XUL [5,14] was presented in 1999 by the Mozilla project to specify Graphical User Interfaces of the Mozilla-browser in platform-independent matter. XUL allows the specification of interactive objects like buttons, labels, and text fields. We can find these objects in tools for creating GUI's like Java.AWT and Java.Swing.

Based on an existing project for eclipse a GUI editor for XUL was developed [27]. It was built in such a way that co-operation wit task models and generated user interfaces became possible. The following example will demonstrate how this editor can be incorporated into the development process of interactive software.

EXAMPLE OF A DEVELOPMENT PROCESS OF A USER INTERFACE

To demonstrate the process of developing applications with our tool-set, we would like to show standard example, which is a mail-managing system. This system is able to manage received mails, i.e. to store them and display at request, and also to send new mails.

Initial Task Model

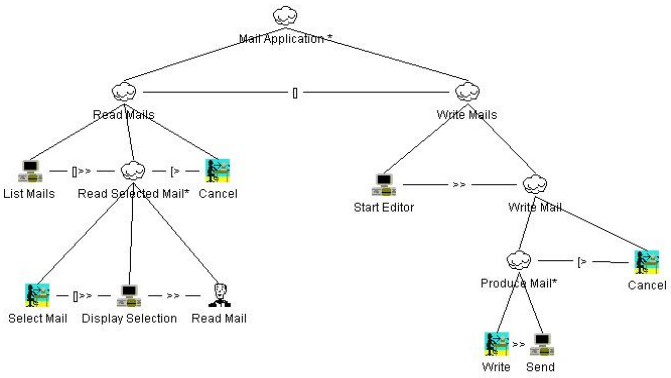

Figure 2 – Initial Task Model

Initial requirements engineering might have resulted in the CTTE-task model represented in Figure 2. According to this model a user may either read his mail, or write a new one. To read his mails, he has to select a specific mail from a list that is generated and presented to him by the application. Once he has selected a mail he gets its content displayed. Select and display are consecutive subtasks of an iterative tasks that can be cancelled at any time.

Writing mails is modelled in a similar manner. After a user decides to write a mail he has to enter the iterative task produce mail, where he is requested to compose a new mail and, after finishing this, the application will send it away. This sub-task may also be cancelled at any time.

From Task Models to Dialog Graphs

While there is currently no satisfying way of an automatic generation of dialog graphs, our tool "DiaTask" (see e.g. [18],[4]) can be used to create them manually.

By using DiaTask, a designer at first has to decide, how many views are desired, and whether each of them is modal, single or multiple. Next step is to assign relevant tasks to views. The underlying task model determines the set of tasks, which can be distributed on views. Thereafter the designer has to model transitions between tasks and views. DiaTask does support necessary operations to do this.

For this example a decision was made to use 4 different views. A start screen, where a user decides whether he wants to read or write a mail, a single view for reading mails, a multi view for writing a mail and a modal dialog was designed to visualize progress during send operations.

The resulting dialogue-graph is shown in Figure 3. One might notice that no task is attached to the fourth view (send progress). It was added due to a technical design decision to have this window visualizing the progress of sending a mail. There was no task in the task model forcing to have this view.

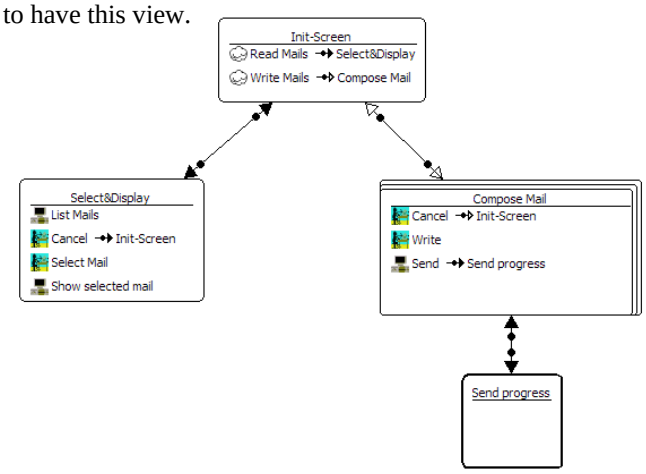

Figure 3 –Dialog graph for sample application

From Dialogue Graphs to Abstract Prototypes of User Interfaces

On a given a dialog graph DiaTask can generate an initial abstract prototype in a WIMP style that mainly reflects the navigation structure of the user interface. Windows are used to represent views and elements of the views are mapped to buttons as can be seen in Figure 4 for the example dialog graph of Figure 3. Other task-element mappings can be achieved by applying a different presentation model.

This generated AUI is stored in XUL format. It is possible to animate the designed dialog graph, for example for testing purposes. When animating, DiaTask uses the before generated XUL to have a graphical representation for each task.

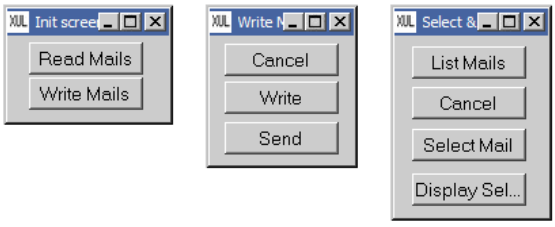

Figure 4 - Pre-Generated Views

Beside its original purpose to carry layout information, each XUL element contains control attributes. Of special relevance is an identification attribute (id) of the associated task. This id is generated by DiaTask while editing the dialog graph and is unique for each task. It allows a trace back of interactions on the final graphical user interface to the corresponding task. Other attributes beside "id" are used to mark and track changes in task-view assignments.

From Abstract Prototypes to Concrete GUIs

Following the generation of abstracts user interfaces (AUI) for testing the dialog structure; in a next step concrete user interface (CUI) is to be designed.

We can support this step with our XUL editing tool (XUL-E) [27]. It was developed as a plug-in for the rich-client platform "Eclipse" [69]. Beside its graphical editing features its main purpose is to support our evolutionary approach. For that some information exchange between XUL-E and DiaTask is necessary. This is handled by a slightly enhanced version of the XUL language, which is called XULM. Enhancements include the possibility to

- store multiple views/windows in one file,
- define a repository for pre-designed components
- define placeholders that refer to these components and
- other pattern related meta-information

The editor XUL-E itself is under continuing development, and is currently able to edit most layout-affecting XUL elements.

XUL-E uses DiaTask's generated AUI as starting point for layout refinements. The basic idea of an integrated editing process, as presented here, is to edit by replacements. To design the user interface for a certain task, one can replace its current visualization by another one.

A user interface designer has to proceed in the following way: At first he has to select a visualization (graphical element) of a task (e.g. a button), choose "replace", and select via "drag & drop" a graphical element or a pre-designed component that replaces the original one. Proceeding in this way makes it possible to maintain any task-related attribution of an element and accordingly keep connections to the task model.

In the example application we now replace buttons in the abstract user interface specification by other more appropriate ui-elements. In view number 2 e.g. the button "List Mails" is replaced by a tree structure and a list box, whereas the button "Display selected Mail" is replaced by three labels and some text displayers. "Select Mail" is exchanged by an "OK_Help_Cancel"-component (see Figure 5).

Replacing a single element, like a button for example, by a more complex component, raises the problem of where to attach task-related information to. While it would be possible to actually apply these data to every element of a complex component, this is probably not the desired behaviour. Also it is imaginable that components contain visualization stubs for referencing other than the current task. Therefore XULM offers fine-grained control on this matter.

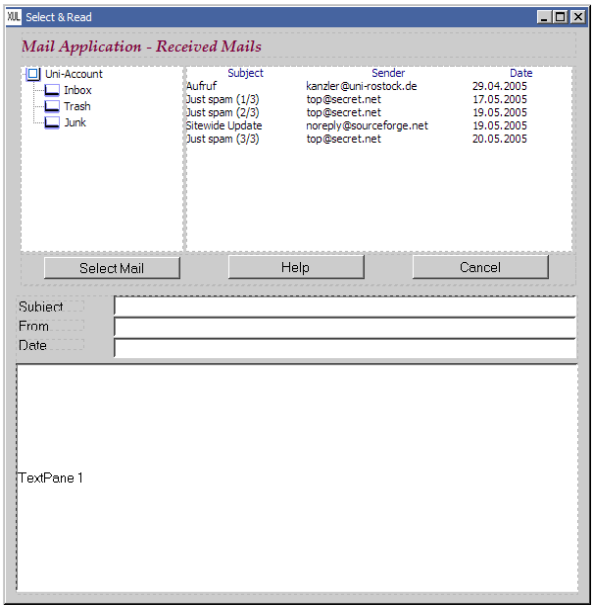

Figure 5 - Designed GUI for Select & Read

An element inside a pre-designed component may be marked whether it should have task-control-data applied or not. The default assumption is that they have not to be applied (value = "none"). A second option is "implicit", meaning to apply any task-control-data of the replaced element. As a third option the value "explicit" can be defined, to define that the marked element in this replacement requires user interaction. A user has to decide, whether to delete this element, as it is currently not needed or to set manually, which task this element belongs to. In this case, as consequence it might be necessary to adapt the underlying dialog graph and even the task model. Both cases (delete/manual) require a consistency check. The tool DiaTask is more appropriate for this purpose than XUL-E, therefore those checks are done there.

For logical and organizational reasons XULM offers to group components into packages. Those packages again can contain packages, creating a hierarchy in this way. Packages are stored in repositories – currently XULM-files – which can be referenced from any other XULM file and are dynamically loaded by XUL-E or any other tool that makes use of XUL-E's engine.

As each package can contain multiple components, it is conceivable to group different visualizations for the same task(s) into one (sub-) package. To support this approach, one component of a package may be declared as default component.

Generally a reference is defined in terms of XULM placeholder-elements. XUL-E's engine determines the component that is referenced and inserts the XUL content of this component into the view. By referencing a package, its defined default-component would be used for visualization purposes. Beside this, explicit referencing of a single component is supported too.

Thus XULM provides two ways to adapt a CUI to different contexts-of-use, either by using a different repository or by overriding package defaults.

For the example application, five components were created, which were grouped into two packages. The view "Select & Read" was designed using components as replacements. A component consisting of a tree and a listbox replaced the button of task „List Mails". Button "Select Mail" was exchanged by an "OK_Help_Cancel"-component and button "Display Selection" consists now of three labels and some text displayers. The result is shown in Figure 5. Note that sample texts have been integrated for demonstration purposes only.

After redesigning views, DiaTask can still be used to animate the dialog graph. At this point it is possible to discuss the application based on a concrete design instead of more abstract – button-represented – tasks.

The animation of the dialog graph becomes "more readable" for users and is more appropriate for discussions. Figure 6 shows an animation state, where a user writes a mail. The simulation engine of DiaTask is currently restricted to a subset of XUL-E's capabilities, so there is some difference in the views appearances.

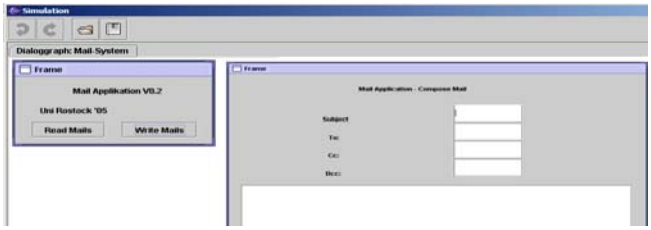

Figure 6 - refined abstract prototype of Fig. 4 in animation mode

EVOLUTION OF MODEL

On an animated walk-through different problems of the application could be identified. Examples are missing functionality that has to be added, tasks that better are associated to another view and redundant or unnecessary tasks, which have to be eliminated. These modifications will be executed on task or dialog model level, thus in our approach DiaTask is the tool responsible for that.

While modifying views, DiaTask tracks the individual changes to each views and marks the kind of change by XULM attributes. On view level a change can be an added or removed task, or transitions between tasks were changed.

After editing of task and dialog model is finished, XUL-E will get started to apply the changes to the CUI. XUL-E shows every modified view, and highlights the changes that applied to these.

APPLYING A UI-PATTERN

The outlined approach also enables us to make use of ui-patterns to design the CUI. To integrate patterns into XUL-E, its already presented concept of packages and components is used. Pattern instances are pre-arranged as components and stored in a repository, they are applied to view elements by replacing just as any other component.

In following section we enhance our sample application by using a user-interface pattern. We will apply the "wizard"-pattern to our application.

The "wizard" pattern requires constructing a navigation structure that guides a user step by step through every necessary operation to achieve a specific goal. We are going to use a simple instance of this pattern that guides a user through entering text properties, one per page, and allows him to call specific context help on each property. In context of our example we will use it for entering administrative data to access a user's mail account.

At first we have to extend our task model with specific tasks for entering each property, e.g. "Enter Name" and "Enter E-Mail". A possible resulting task model is presented in figure 7.

Secondly we use DiaTask to add a view that will be the start view of our pattern usage. All "enter …"-properties are assigned to this view. Also a transition from the start view to our newly created view is added. Note that this last modification is marked via XULM. XUL-E will force a designer to apply this change on CUI level, i.e. redesign this view to incorporate this new task. After modifications are finished the dialogue model of figure 8 might be a result. With it, DiaTask creates a first prototype of our property-entering dialog, consisting of buttons, similar to that of figure 4.

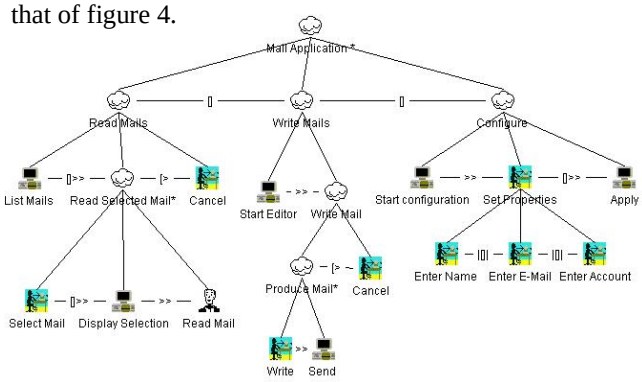

Figure 7 – Enhanced task model, including property input tasks

Next, XUL-E is started to edit the new view, and of course as already mentioned the start view. We now want to apply "wizard" pattern to these buttons and with that to replace each of these buttons with a single view containing a text input field and let these created views be interconnected by transitions to the respective predecessor, successor and the last of created views, which should be the "apply"-view.

To achieve this we use a pre-designed component that contains an instance of "wizard" that visualizes every single task in above described manner. Details on the construction of such a component will follow.

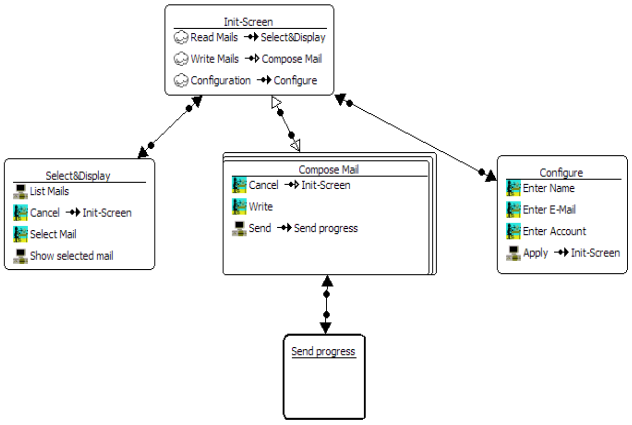

Figure 8 – Dialogue graph including property input

A designer now has to select all the tasks he wishes the pattern instance applied to, and he has to do this in the same sequence in which he wishes the separate pages of the wizard to appear. For our example we chose the sequence: "Name", "Email address", "Account" and "Apply". Now as last step all these selected tasks are replaced with the visualization defined within the replacing component.

In principle these replacements follow the already mentioned procedure for replacing a view element with another one. Differing procedures are covered in greater detail in the following section.

The result of our replacement is 4 separate views, whose transitions between each other are already defined on XULM level. Figure 9 shows the effect of the transformation to our applications dialog model.

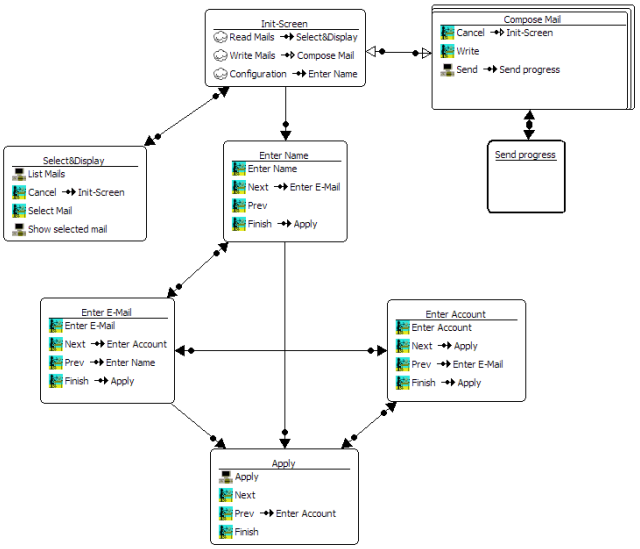

Figure 9 –Dialogue graph after pattern transformation

One can see that view “apply” does not behave as necessary for the application to run. Therefore manual refinement is needed. The view’s task “Next” probably would be removed, and for “Finish” a transition to “Apply” defined. Transition for task “Apply” should be back to view “Init Screen”.

After these fixes are done, each view can be individually designed as described earlier, including further replacements or the appliance of other patterns.

At the end of this “transform – refine – design” cycle, the application prototype can still be animated, e.g. to verify and test the effects of pattern application. All tasks and the related GUI elements have still their association to the application’s task model.

DEFINING A PATTERN INSTANCE AS COMPONENT

In the previous section we used an already existing component that implemented a “wizard” pattern. In the following is outlined how such a component is defined.

A pattern instance, in our approach, consists of a basic layout that is created using XUL and has task-control data that defines which elements get what kind of task-data applied. Furthermore a component can have sub dialog models assigned, which get integrated into an applications dialog model by applying the component, currently this is restricted to very basic internal view transitions.

All these information are stored within XULM files that serve as component databases. XUL-E and DiaTask will access these databases to execute their respective operations.

Just like any other component, pattern instance components (PIC) can make use of placeholders to import other components into their design. Unique to PICs is that they can be used to replace more than one element at once.

On such multiple replacements it is possible to access and use task-data of each belonging task separately. Potentially we will also integrate a basic scripting feature to further increase user control over the replacement process.

A PIC definition uses a lightweight description-language and consists of the following:

1. Input:
    a. Parameter type is either single tasks or lists of tasks
    b. Input parameter need to be defined with a name and declare the type they are of
    c. Input parameter can have a short description
2. Operations:
    a. A “foreach”-loop is supported that traverses a tasklist and executes the loop-body as often as there are task in the list; In the n-th loop run, declared loop-variable points to the n-th task in list
    b. An IfDef-Else-EndIf construct may be used to handle unset values
3. Accessible object information:
    a. A loop-variable knows about start, last, previous, next and the current element
    b. Task objects have an id, view (if they are associated to any), name and a transition target
    c. View objects have an id and a name
    d.

The output of a PIC based transformation is always of type XULM.

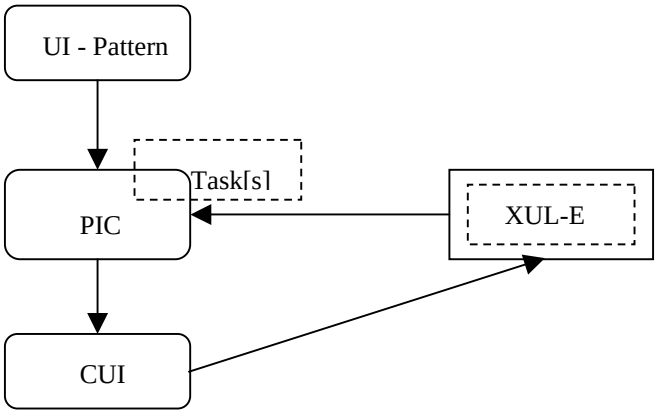

Figure 10 – Visualization of relations between Pattern, PIC and XUL-E

Figure 10 illustrates idea of how we support the use of ui-patterns in our approach. A PIC is an implementation of an abstract pattern, but still cannot be used for visualization. It has to be instantiated; which is initialized with tasks from XUL-E the result of the transformation replaces those tasks.

The definition of the previously used PIC “wizard” is presented in Listing 1. It can be seen that XULM and XUL are mixed in one file. Packages and placeholders have also been used, so it is also an demonstration of these concepts.

For increased readability and to reduce complexity some parts have been left out, e.g. XML-Namespaces and designs for irrelevant components.

```
<guidefinitions>
<components>
  <package name="uipatterns" hasDefault="false">
   <component name="wizard"
       isPatternInstance="true">
    <inputTask paramName="tasklist"
       paramType="taskListType"
       paramShortDescription=
         "Ordered List of Tasks" />
     <opForEach source="tasklist"
        loopPointer="#current">
      <window orient="vertical">
       <hbox>
         ... Logo and Label ...
       </hbox>
       <hbox>
         <label
           processTasks="true"
           value="#current.task.name"
           TASK_DATA="implicit"
          />
          <textbox TASK_DATA="implicit">
               Value here
           </textbox>
          <button value="Help on this matter"
                TASK_DATA="implicit" />
        </hbox>
        <hbox>
         <placeholder
           usepackage="defComp.prev_next_finish"
           defaultvalue="taskaware"
         />
        </hbox>
       </window>
      </opForEach>
     </component>
    </package>
  </package>

  <package name="defComp" hasDefault="false">
   <package name="prev_next_finish"
            defaultvalue="nontaskaware">
    <component name="nontaskaware">
      <!-- plain design of the three buttons -->
    </component>
    <component name="taskaware"
       associatedDialogModel="">
     <hbox>
      <button label="Prev"
        transition="#current.prev.task.view" />
      <button label="Next"
        transition="#current.next.task.view" />
      <button label="Finish"
        transition="#current.last.task.view" />
     </hbox>
    </component>
   </package>
  </package>
 </components>
</guidefinitions>
```

Listing 1 – Definition of PIC "wizard" in a component

CONCLUSION AND FURTHER WORK

We have shown that the combination of the DiaTask and XUL-E tools can support model-based development of user interfaces. Starting with a task model a designer can interactively create and design abstract and concrete user interface for an application. As the resulting specifications for AUI and CUI are in XUL format, there is even a chance of converting them into native code for programming languages by using specialized compilers.

Furthermore we proposed a method for integrating ui-patterns into the overall process. Transformations templates, attributed with tasks, serve as instances of patterns that can interactively be applied to a concrete user interface element or an abstract user interface element. A working example utilizing this approach has been presented.

In future it has to be checked for which groups of ui-patterns our method is appropriate and for which it might not or only with difficulties be usable. An example for the latter group is the "Breadcrumb"-pattern, an instance of it hardly seems to be realizable.

It is conceivable that more complex patterns, or even a more sophisticated "wizard" pattern instance, require enhancements to current scripting possibilities as well as to the current model changing facilities of XUL-E in cooperation with DiaTask, which in both cases cover only basic operations at the moment.

We plan not only to design a pattern interface component and equip it with some script commands, but also to connect it with a complete task or at least dialog model. On applying such a PIC to a task, or a set of tasks, the models would get plugged into the existing models of the application.

Such heavyweight PIC's probably require an own editor, which then also should be created. The next step here would then be to build and maintain a PIC library.

Also another approach for combining DiaTask and XUL-E might be worth some testing. DiaTask could be used to assign stereotypes, i.e. desired patterns, to certain views or group of tasks. It afterwards could call XUL-E's engine to convert these marked views into instances of the selected pattern. This would also result in an AUI with patterns applied.

A technical issue is to choose another script language, for example Velocity [25], instead of creating and maintaining an own one.

REFERENCES

1. Cameleon: http://giove.cnuce.cnr.it/cameleon.html.
2. Clerxkx, T.; Luyten K.; Conix, K.: The Mapping Problem Back and Forth: Customizing Dynamic Models while preserving Consitency, Proc. TAMODIA 2004, P. 33-42.
3. Constantine L.L: Canonical Abstract Prototypes for Abstract Visual and Interaction Design, in Jorge J. A. et. al (Eds): *Proceedings DSV-IS 2003*, LNCS 2844, Springer Verlag, Berlin, 2003, P. 1-15.
4. CTTE: The ConcurTaskTree Environment. http://giove.cnuce.cnr.it/ctte.html.

5. Deakin, N.: XUL Tutorial. XUL Planet. 2000.
6. Dittmar, A., Forbrig, P.: The Influence of Improved Task Models on Dialogues. *Proc. of CADUI 2004*, Madeira, 2004.
7. Dittmar, A., Forbrig, P., Heftberger, S., Stary, C.: Tool Support for Task Modelling – A Constructive Exploration. *Proc. EHCI-DSVIS'04*, 2004.
8. Dittmar, A., Forbrig, P., Reichart, D.: Model-based Development of Nomadic Applications. In *Proc. of 4th International Workshop on Mobile Computing*, Rostock, Germany, 2003.
9. Eclipse: http://www.eclipse.org.
10. Elwert, T., Schlungbaum, E.: Dialogue Graphs – A Formal and Visual Specification Technique for Dialogue Modelling. In Siddiqi, J.I., Roast, C.R. (ed.) *Formal Aspects of the Human Computer Interface*, Springer Verlag, 1996.
11. Limbourg, Q., Vanderdonckt, J.: Addressing the Mapping Problem in User Interface Design with USIXML, *Proc TAMODIA 2004*, Prague, P. 155-164
12. López-Jaquero, V.; Montero, F. ; Molina, J.,P.; González, P.: A Seamless Development Process of Adaptive User Interfaces Explicitly Based on Usability Properties, *Proc. EHCI-DSVIS'04*, p. 372-389, 2004.
13. Luyten, K., Clerckx, T., Coninx, K., Vanderdonckt, J.: Derivation of a dialog model from a task model by activity chain extraction. In Jorge, J., Nunes, N.J., e Cunha, J.F. (ed.), *Proc. of DSV-IS 2003*, LNCS 2844, Springer, 2003.
14. Mozilla.org: XUL Programmer's Reference 2001.
15. Paterno, F.; Mancini, C.; Meniconi, S: ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models, *Proc. Interact 97*, Sydney, Chapman & Hall, p362-369, 1997.
16. Paterno, F., Santoro, C.: One Model, Many Interfaces. In Proc. of the Fourth International Conference on Computer-Aided Design of User Interfaces, p. 143-154. Kluwer Academics Publishers, 2002.
17. Puerta, A.R. and Eisenstein, J.: Towards a General Computational Framework for Model-Based Interface Development Systems. Proc. of the 4th ACM Conf. On Intelligent User Interfaces IUI'99 (Los Angeles, 5-8 January 1999). ACM Press, New York (1999), 171–178
18. Reichart, D.; Forbrig, P.; Dittmar, A.: Task Models as Basis for Requirements Engineering and Software Execution, Proc. of. Tamodia 2004, p. 51-58
19. Sinnig, D., Gaffar, A., Reichart, D., Forbrig, P., Seffah, A.: Patterns in Model-Based Engineering, *Proc. of CADUI 2004*, Madeira, 2004.
20. Sinnig, D., Javahery, H., Forbrig, P. and Seffah, A., "Patterns and Components for Enhancing Reusability and Systematic UI Development", in Proceedings of HCI International, Las Vegas, USA, 2005.
21. Teuber, C.; Forbrig, P.: Modeling Patterns for Task Models, Proc. of Tamodia 2004, Prague, Czech. Republic, p. 91-98.
22. TERESA: http://giove.cnuce.cnr.it/teresa.html
23. UIML Tutorial, http://www.harmonia.com
24. UsiXML: http://www.usixml.org/
25. Velocity: http://jakarta.apache.org/velocity/
26. Wilson, S.; Johnson, P.: Bridging the generation gap: From work tasks to user interface design, In Vanderdonckt, J. (Ed.), *Proc. of CADUI 96*, Presses Universitaires de Namur, 199, p. 77-94.
27. Wolff, Andreas: Ein Konzept zur Integration von Aufgabenmodellen in das GUI-Design , Master Thesis, University of Rostock, 2004.
28. Wolff, A.; Forbrig, P.; Dittmar, A.; Reichart, D.: Linking GUI Elements to Tasks – Supporting an Evolutionary Design Process, Proc. of. Tamodia 2005, Gdansk, Poland, p. 27-34
29. XIML: http://www.ximl.org
30. Paquette, D.; Schneider, K.: Interaction Templates for Constructing User Interfaces from Task Models, Proc. Of CADUI 2004, Madeira Island, Portugal, p. 221-232

From Requirements Analysis to Architecture Evaluation of a Ubiquitous Multimodal Multimedia Computing System

Manolo Dulva Hina
LATIS[1] and PRISM[2] Laboratories
[1]École de technologie supérieure
[2]Université de Versailles-St.-Quentin-en-Yvelines
1-514-890-1092
manolo-dulva.hina.1@ens.etsmtl.ca

Chakib Tadj
École de technologie superieure
1100, rue Notre-Dame Ouest
Montréal Quebec H3C 1K3 Canada
1-514-396-8555
ctadj@ele.etsmtl.ca

Amar Ramdane-Cherif
Université de Versailles-St.-Quentin-en-Yvelines
45, avenue des États-Unis
78035 Versailles Cedex, France
33-1-39.25.43.12
rca@prism.uvsq.fr

ABSTRACT
Our ubiquitous multimodal multimedia (MM) computing system selects the appropriate media and modalities based on the user's context and user's profile. The overall user context is decided based on four parameters, namely the user's location, the noise level in the user's workplace and the presence or absence of other people in the user's workplace (a.k.a. safety factor), and the computing device used by the user. The user's profile identifies if the user is a regular user or a handicapped. The user's handicap determines if the user is manually handicapped, visually impaired, a deaf or a mute.

Machine Learning (ML) is concerned with the development of techniques allowing the computer to acquire knowledge. In our work, a ML component resolves all questions related to the system's selection of media and modalities with reference to the user's situation. This ML component uses a priori training sets which contain records of scenarios. Every scenario record is composed of a pre-condition scenario (i.e. the user context), and its corresponding post-condition scenario (i.e. the media and modalities that are appropriate for such context).

Given a certain context, the media and modalities listed in the post-condition scenario are set for activation. A problem arises when a media or modality is found missing or defective which could potentially cause the system to stall or to crash. Our system uses its acquired knowledge to find a replacement to the defective component. To do so, the ML agent consults its knowledge database which contains a list of replacements to a failed component. If the list is empty or when the selected device/modality and all its replacements have all failed, the ML system is trained; each training yields one device included in the list. The more the ML system is trained, the more resilient the system becomes over components failure.

This paper demonstrates the design of a fault-tolerant ubiquitous MM computing system. The requirements analysis is undertaken by considering the quality attributes desired by different stakeholders. We use the attribute-driven design method in the requirement analysis and Architecture Tradeoffs Analysis Method in evaluating the system architecture.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Domain-specific Architecture – *requirements analysis, attribute-driven design, quality attributes, software architecture, Architecture Tradeoffs Analysis Method (ATAM).*

General Terms
Documentation, Design.

Keywords
Multimodal multimedia system, incremental learning, activity-driven design, quality attributes, software architecture, ATAM.

1. INTRODUCTION
A multimodal user interface allows user to do computing with more than one mode of interaction. Indeed, incorporating *multimodality* into a computing system makes it more accessible to a wider range of users, including those with impairments. A MM system in [1] combines two different types of data – one coming from a multimodal source, usually demonstrated by human action (e.g. speech, pointing object using an electronic glove), and another coming from the usual media (e.g. keyboard, mouse) – and the fusion of these data produces a new data that has a completely different meaning to the system. Our work in [2] selects the media and modalities that are deemed appropriate to the user's situation. The user's situation is a function of the user's context and profile, and the user's current environment and hardware profile. To do this selection, the system adapts ML process. Indeed, there is one system component that is responsible for the resolution of any question concerning the media and modalities selection. The knowledge acquisition of the ML system is incremental – it is trained to new scenario and to new media or modality component. The more it is trained, the more it becomes reliable over different conditions. This paper is all about the design of this system, from its requirement analysis to the evaluation of its software architecture.

A ubiquitous system allows the user to continue working on an interrupted task anytime and anywhere. In our work in [2], the user's task and profile as well as machine knowledge are all made transportable form one computing environment to another, basically "following" wherever the user goes. To realize ubiquity, support for wired, wireless and mobile computing is imperative.

2. RELATED WORK

The concepts of *distributed computing* merged with wired and wireless or *mobile computing* were the foundations of ubiquitous computing, sometimes known as *pervasive computing* [3]. Satyanarayanan's work on the concepts of *mobile information access* [4] and *Coda* [5] are important in pervasive computing, and so does that of wireless networks [6]. Multimodal multimedia computing [1] is about the fusion of data coming from the usual media (i.e. keyboard, mouse), and modality that uses human action (i.e. speech). Our ubiquitous MM computing system (UMMCS) however is not involved in the fusion of these data; instead ours selects the media and modality based on user's context and user special needs. Software architecture [7-9] is the blueprint of the system that is to be built. Indeed, the essential quality attributes could be viewed, analyzed and synthesized in such blueprint even before building one. Important works related to the achievement of the essential system quality through software architecture include [10] and [11, 12]. The architecture style or pattern [13] is also a factor in deciding which quality attribute would be prioritized. In this paper, our architectural framework along with detailed parts of the complete system architecture is presented. A specific quality attribute that is essential to the system is presented along with the stimulus that could affect the system from obtaining such quality.

Our UMMCS is based on machine's knowledge acquisition [14]. Its learning acquisition is progressive. Related work in machine learning (ML) include [15], Learn Sesame [16], and conversational agent [16]. Unlike that in [17], our work's self-repairing system through dynamic reconfiguration (DR) [18] is incremental ML-related and so the system's capacity to do self-repair is a function of its acquired knowledge through training. The more the system is trained, the smarter it has become.

This paper illustrates our UMMCS using the basic software engineering practices – requirements analysis, system modeling, architecture design and architectural evaluation.

3. SYSTEM REQUIREMENTS ANALYSIS

In this section, we present the beginning and the evolution of our research work. Unlike the usual software project wherein a specific client with specific needs would request a designer for a system design that would realize the client's needs, ours is based on a classic model of an academic research work wherein the student does most of the analysis, design and validation, and the thesis supervisors act as the clients and stakeholders.

3.1 Statement of the Problem

The work by Djenidi et al in [1] considers the fusion of inputs from a usual media (e.g. keyboard, mouse, etc.) and modalities (e.g. speech, eye gaze, etc). Their combination produces a new set of data that has different meaning from depending on whether these inputs arrive at the same time, one after the other, etc.

In the research work mentioned above, the media and modality for fusion are already identified. Indeed, a new problem arises from such system. The following is the research question that needs some clear answers:

- *Given a certain computing environment, how can we select the media and modalities so that we can realize the fusion of their inputs?*

Indeed, our new work is no longer in the fusion of media and modalities input but rather on their correct selection based on user's computing situation. It is necessary to consider that one specific media may not be present or is found defective in a certain environment, and another modality could be completely irrelevant in a certain situation, such as the speech recognition in a noisy environment. An assumption that the needed media and modalities are both present and functional or appropriate in a any computing environment is wrong. Therefore, a scientific approach for the selection of media and modalities based on the user's computing situation is needed.

3.2 System Requirements

There is a need to create a computing system that correctly selects the media and modalities based on the computing environment itself. The following are the analysis of the system requirements:

1. The media and modalities should be selected based on user's context. Among others, the context measure should consider the location of the user and the noise level of the user's environment. The security of the user's workplace should also be taken into account.

2. Given the same context parameters mentioned above, the media and modalities that are appropriate for a regular user are completely different from the ones that are appropriate for the disabled users (i.e. blind, deaf, etc.). Indeed, the user's special needs should be taken into consideration as well.

3. Given that a mobile user would be using different computing environment and therefore different computing device (e.g. PC, laptop, PDA, etc.), the media and modalities to be selected must be appropriate for the computing device.

4. Considering that there are so many possible combinations of user's context, user's special needs, and user's computing device, with each combination yields its media and modalities selection, then the system to be built must "remember" each of these "scenarios" so that when the same scenario happens again in the future, the system could "act" accordingly and intelligently.

5. For each scenario mentioned above, the media and modalities that are selected for the scenario may or may not be available, and may or may not be functional. In the event that a device (i.e. a media or a modality) is missing or is found defective, then the system should find its replacement if it intends to remain fault-tolerant.

3.3 Requirements Definition

The system requirements mentioned above should be further defined for clarity purposes. The following are therefore the detailed definitions of the system requirements:

1. *User's Context* – it refers to the overall assessment of user's situation which is based upon: (i) user's location, (ii) the noise level in the user's workplace, and (iii) the security factor which is the detected presence or absence of other people in the user's workplace.

2. *User's Special Needs* – this should be one element of every user's profile. It would identify if the user is a *regular user* or a *handicapped*. If handicapped, then the system should be informed if the user is (i) *manually disabled*, (ii) *visually*

impaired, (iii) a *deaf*, or (iv) a *mute*. In the event that the user has more than one disability, then the media and modalities that are appropriate for his case is the intersection of the media and modalities that are appropriate for each individual disability.

3. *The Computing Device* – this essentially identifies the type of computer the user is utilizing. This could be a PC, a laptop or a PDA.

4. *Scenario* – Every user scenario should reflect the cause and effect combination, otherwise known as the *pre-condition scenario* and the *post-condition scenario*, respectively. Hence, the pre-condition scenario would be the combination of the user's context, special needs and computing device, and the corresponding post-condition scenario would be the media and modalities that are appropriate for the scenario.

5. *Machine Learning* – this refers to the machine's acquisition of knowledge. The knowledge to be acquired is the recognition of the user's computing situation (i.e. pre-condition scenario) and the selection of appropriate media and modalities (i.e. post-condition scenario). This knowledge has to be stored in a repository (i.e. knowledge database) that could be accessed wherever and whenever the user needs it. The learning system should also consider a list of replacements for any failed device in any given context.

The following subsection would provide further details for each of these requirements.

3.3.1 The User Context

The details of each user's context parameters are as follows:

1. *The user location* – this refers the detection of the user's whereabouts. We employ the use of a *global position system* (GPS) attached to the USB port of the computer being used by the user. There must be a software agent that should take sample GPS readings (1 sample per minute); after 5 samples, the agent yields a final result confirming the location of the user. At the end, the agent would conclude if the user is (i) at *home*, (ii) at *work*, or (iii) *on the go*, that is neither at home nor at work (e.g. in the cafeteria, in a park).
2. *The noise level* – To measure the noise level in the user's workplace, there needs to be a sampling device that measures the noise level and a software agent that collects the samples. After 5 samples (i.e. 1 sample per minute), the agent should conclude if the user's workplace is (i) *quiet*, (ii) *acceptable*, or (iii) *noisy*. The device used to measure the noise is the BAPPU measuring device (www.bappu.com). A noise sample could be any of the following: (i) *40 decibels (dB) or less = quiet*, (ii) *41 to 50 dB = acceptable*, and (iii) *51 dB or more = noisy*. To accommodate every user's perception of noise, a user interface should be added in the design so that a user could have the means to enter this own thresholds to suit his noise perception.
3. *The safety factor* – this refers to the detection of how safe (or risky) the user's workplace is based on (1) the presence or absence of other people in the vicinity of user's workplace, and (2) who is sitting in the user's chair facing the computer. To accomplish this, a software agent has to read samples from two sensors, namely: (1) an *infrared detector* that detects the presence of other people within the vicinity of user's workplace, and (2) a *camera with retinal recognition* that detects if it is the legitimate user who is sitting in the user's seat. The results of these two sensors are combined together to determine the safety factor in the user's workplace. The agent yields a *final assessment* that indicates if the safety factor is either (i) *good or ideal*, (ii) *acceptable*, (iii) *sensitive*, or (iv) *bad, worse or worst*. The calculation for the final assessment will be provided in the next section.

3.3.2 The User Profile

The *user profile* (UP) will be a record containing vital computing information about the user. Every user should have a UP. When a *new user* is added to the system, the system creates a *new UP* for him. In general, the UP should be *dynamic*; the user can modify its contents whenever he wishes. The UP should be *ubiquitous* in order to support the needs of a mobile user; hence this *private* data structure is omnipresent, basically following the user wherever he goes. For the purpose of initial system design, the UP is composed of two parts, namely:

1. The *user profile* – this contains the user's username, password and the list of computing units the user utilizes including their identifications (i.e. IP addresses) and their corresponding schedules. Note that, in a ubiquitous and pervasive environment, the user is moving from one environment to another, and so we want to keep track of his location via this part of the UP.
2. The *user's special needs* – if applicable, the user has to indicate the nature of his disability. This is necessary so that the system could correctly select the media/modalities that are appropriate for his situation. The default value is that the user is a regular, non-disabled user.

3.3.3 The Machine Learning Process

The *machine learning* (ML) component of the system would acquire knowledge on all possible combinations of user context, user computing device and user's special needs (also known as pre-condition scenarios) and for each combination would select the appropriate media and modalities (the post-condition scenario). There is an a priori training set which shall be the initial knowledge that the machine learns. As such, the machine's intelligence shall be limited to this set. For the machine to evolve, its machine learning should be incremental, that is, one at a time, on a situation by situation basis. For as long as there is something new to learn, an incremental ML system continues its progressive acquisition of knowledge.

Our ML system must also have its list of replacements to every failed/missing device in any given context. Also, it must have knowledge to evaluate the suitability of every media or modality to any given context. Our work in [19] provides details to the design of the incremental learning component of this system.

3.4 Modeling System Requirements

To model the system requirements, we use the Data Flow Diagram (DFD) model which shows data processing as the data flows through the system. The modeling is seen from the system's functional perspective. Figure 1 shows the level 0 of the DFD of our system. Level 0 of the DFD demonstrates the ubiquitous MM computing system (UMMCS) with all the inputs coming into the

system and all the outputs produced by the system, and some data repositories also indicated. In general, level 0 is too general that the system being developed or modeled appears just as a nebulous concept. Further details should be provided by designing extra levels (i.e. level 1 up to level n) until system details are clear enough for the interested stakeholders. Due to space limitations, we could only show DFD up to level 1. Figure 2 demonstrates level 1 of the DFD. The UMMCS is now partitioned into three main components, namely: the Task Manager Agent (TMA), the Context Manager Agent (CMA), and the History and Knowledge-base Agent (HKA). Their functionalities are as follows:

- *The Task Manager Agent* (TMA) – it is responsible for the management of user's task and profile.
- *The Context Manager Agent* (CMA) – it is responsible for the detection of user's context and eventually the selection of the appropriate media and modalities based on that context
- *Knowledge History-base Agent* (KHA) – it is the component that handles the ML process, including its training and the management of the *knowledge database* (KD).

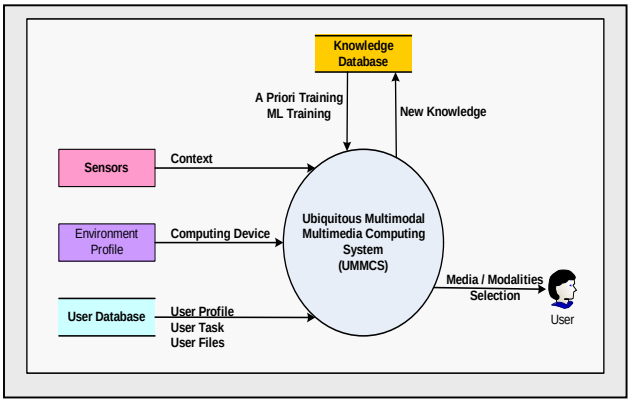

Figure 1. Data Flow Diagram, Level 0 of the UMMCS.

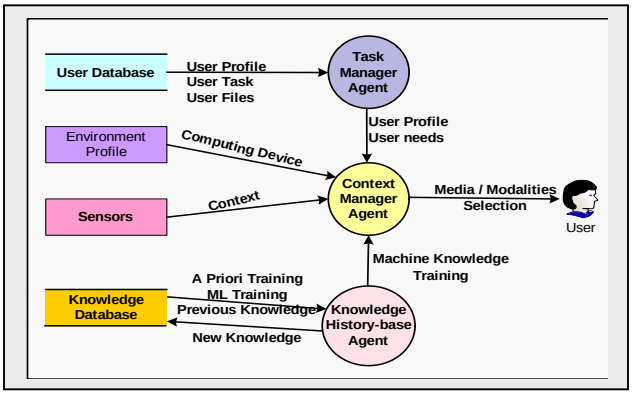

Figure 2. Data Flow Diagram, Level 1 of the UMMCS.

4. ATTRIBUTE-DRIVEN ARCHITECTURAL DESIGN

Software architecture is the structure of the components of a program or a system, their interrelationships, and the principles and guidelines governing their design and evolution over time. The architectural framework in Figure 3 provides some idea of the main components that comprise the system. This framework is transformed into an architecture that satisfies system quality and functional requirements. We use the *attribute-driven design* (ADD) [20] methodology because our architectural design is aimed at achieving the system's desired quality attributes. The ADD steps are as follows:

1. Choose the module to decompose
2. Refine the module according to these steps:
   a. Choose the architectural drivers from the set of concrete quality scenarios and functional requirements. This step determines what is important for this decomposition.
   b. Choose an architectural pattern that satisfies the architectural drivers. Create the pattern based on the tactic that can be used to achieve the drivers. Identify the modules required to implement the tactics.
   c. Instantiate modules and allocate functionalities from the use cases and represent using multiple views.
   d. Define interfaces of the child modules.
   e. Verify and refine use cases and quality scenario and make them constraints for the child modules. This step verifies that nothing important was forgotten.
3. Repeat the steps above for every module that needs decomposition.

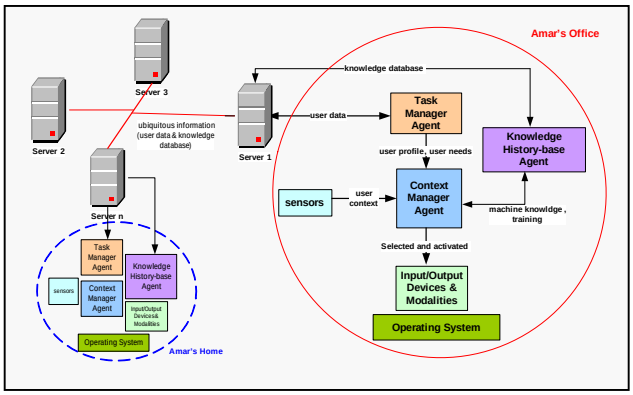

Figure 3. The UMMCS architectural framework.

Here, we briefly describe the essence of ADD. The module decomposition always begins with the whole system. Figure 4 demonstrates the transition from the general framework into a top level decomposition. In Figure 4, the decomposition yields more detailed components that interact with CMA; the sensors in Figure 3 have been replaced by three sets of different sensors, each is managed by its respective agent to produce an individual context. The module decomposition process using ADD is repeated until the architecture is detailed enough to see that it is capable of withstanding all sorts of stimuli that could challenge the system from achieving and maintaining its quality attributes. To do so, each of the system stimuli (i.e. a stimulus is an event that arrives, such as virus attack, that could make the system unstable if not properly attended) is subjected to quality attribute scenario. In general, we make every effort to make our system able to withstand all sorts of stimuli, including the possible breakdown of a sensor or a media or modality, and provide a contingency system action when such a failure occurs. In effect, doing so makes our system fault-tolerant. The architecture shown

in Figure 4 is still far from showing the details of the complete system. Our work in [21] gives more details about software architectural design considerations.

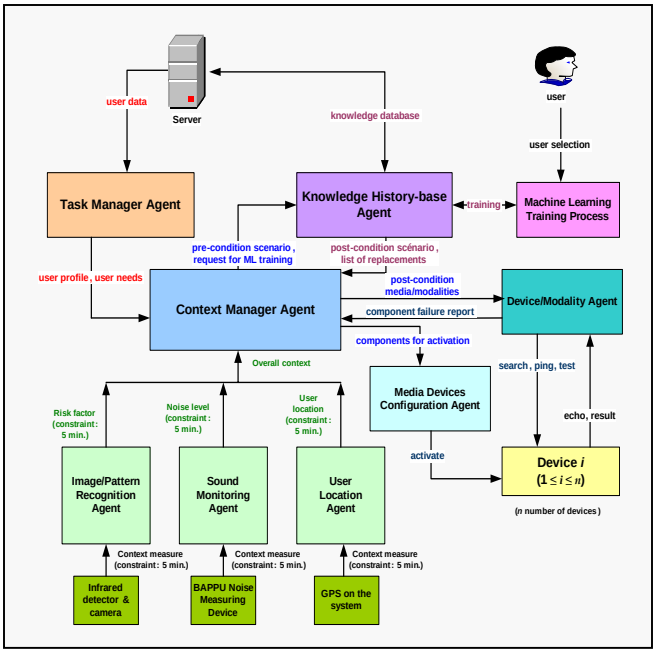

Figure 4. The UMMCS after ADD first-level decomposition.

4.1 Quality Attributes and Attribute-dependent Scenarios

Quality attributes (see Figure 5) are the functional characteristics that the system being designed must possess. The software architecture must demonstrate how these attributes could be achieved. In general, some of the basic qualities that the system must possess (i.e. quality attributes) are the following: (i) availability, (ii) modifiability, (iii) performance, (iv) security, (v) testability, and (vi) usability. *Availability* is about techniques of diagnosing, solving and preventing software or hardware faults and failures. *Modifiability* is about installing change to the system, and the questions of how, when and who will carry this out. *Performance* is making sure that the system is able to respond accordingly when an event occurs. *Security* is about protecting the system and its users and finding ways to deny access to unauthorized users. *Testability* refers to that quality of the system where the designers could easily detect fault as they envisioned it to be. *Usability* is the capacity of the system to do its task with ease, and the kind of support it provides to the users.

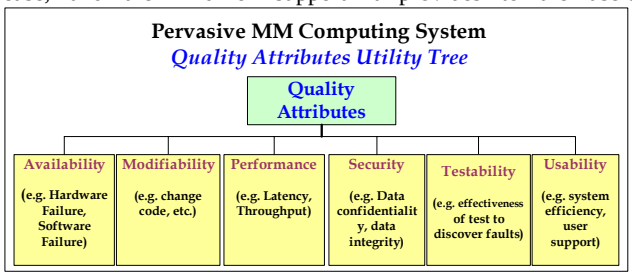

Figure 5. Some basic system quality attributes.

For each of these quality attributes, there is a need to demonstrate how the system would be able achieve such quality attribute. Indeed, the system has to be subjected to a scenario that tests if indeed the system is well designed to achieve each desired quality attribute. This scenario is called the *quality attribute scenario* (QAS). Hence, if the system is to achieve the 6 basic quality attributes, it needs to be subjected to 6 different quality attribute scenarios. Before doing so, the six elements of a QAS are first described below:

(1) *The stimulus*. A stimulus refers to an event that arrives to the system and it must be acted upon. In general, a stimulus is an input to a component; there may be two or more stimuli that could arrive at a given time. A stimulus could be as simple as the user's keyboard input or a computer virus that intends to attack the network.

(2) *The source of the stimulus*. Basically, it refers to the source of the input data (i.e. the one that invokes the stimulus). A source of a stimulus could be a user, a program procedure, or a system component.

(3) *The environment*. This is the condition in which the stimulus has occurred. A snapshot of the environment as the stimulus arrives is important in determining how it will respond to the stimulus. For example, the system's reaction with respect to the stimulus is obviously different if it is previously idle than if it is previously overloaded.

(4) *The artifact*. This refers to the object that is affected by the stimulus. The artifact could be as simple as the file, or as complex as a subsystem or the entire Intranet network.

(5) *The response*. It refers to the processing or the activity that is undertaken by the component or by the environment after the arrival of the stimulus. For the user's input, the response could be as simple as calling a procedure to do some calculations. For a virus attack being detected, the response is either protecting the network or killing the virus.

(6) *The response measure*. This refers to the attribute-specific constraint that must be satisfied by the response. For a performance-related response, such as a website being uploaded into the web browser, the response measure is to have this site uploaded in the next 2 seconds.

Table 1 illustrates the QAS for the six basic quality attributes. The table demonstrates general scenarios that could affect the system's quality attributes in general. For example, the availability of the system (i.e. keeping itself functional amidst varying conditions and faults) could be challenged if some components become faulty, hence, faults are the stimulus. The desired response of the system is to keep itself functional, probably by disabling the defective component. The constraint of this kind response is time. That is, how long could it stand to keep itself functional and not able to stall or crash? If the response is making the defective component functional, then the constraint of the response is the repair time of the faulty component. In an interactive environment, for example, the system designer should know that a component repair time of several minutes is unacceptable to the end user.

In general, the QAS in Table 1 illustrates stimulus considerations that could challenge the system in achieving modifiability, performance, security, testability and usability. Knowing the

stimulus and the artifact affected by the stimulus, the system designer could adapt certain tactics and methods so that the system response to the stimulus is effective and efficient. Table 2 lists down some of the tactics to achieve system quality attributes. Within a tactics, there are two or more methods to choose from; designer must choose one that best fits his system.

Table 1. The quality attribute scenarios.

No.	Quality Attribute	Stimulus	Source of stimulus	Environment	Artifact	Response	Response Measure
1	Availability	Fault such as omission, crash, timing, response	Internal/external to the system	Normal operation, or degraded mode (i.e. fewer features, fall back solution)	CPU, communication channels, persistent storage, processes	Detect event and notify parties, and disable source of event, and continue operation	Availability time, and repair time
2	Modifiability	Modification to a functionality, a quality attribute or system capacity	The user, the developer or the system administrator	Could be at runtime, compile time, build time or design time	The object of modification could be the user interface, platform, environment, or system modules	Undergo modification without affecting other functionalities, and then test, modify and deploy	The cost of modification in general (e.g. money, time, number of elements affected, effort)
3	Performance	Events (i.e. transactions) that arrive in the system	Different sources, which could include even the system itself	The system mode in general (i.e. normal mode or overloaded mode)	The system	Attend to the stimulus, change the level of service	Latency, deadline, throughput, data loss, miss rate
4	Security	Attack from intruder, attempt to access the system from authorized and unauthorized users	Intruder, attacker or legitimate users	The system, either online or offline, connected or disconnected or firewalled or not.	System services and data within the system	Services to legitimate users, system protection from attackers and illegitimate users	The time and resources needed to allow access to legitimate users, and to protect the system from attacks and intruders
5	Testability	Analysis of design, architecture, and components to detect faults	Various stakeholders, including the developers and users	The system being tested at design time, development time, compile time and deployment time	The design piece, a piece of code or subprogram, and the complete application itself	Provide access to test values, module variables and environment test	Probability of failure in case of faults, time to perform test, percent of executable statements executed.
6	Usability	Invoke to use the system and learn its features, getting used to it, minimize impact of error, or adapt to the system	The end user.	The system at runtime or configuration time	The system	System response to user's invocation.	Task time, number of errors, number of problems solved, user satisfaction

Table 2. Tactics /methods to achieve system quality attributes.

Quality Attribute	Tactics	Methods
Availability	Fault Detection	(1) Ping/Echo, (2) Heartbeat, (3) Exception
	Fault Recovery	(1) Voting, (2) Active Redundancy, (3) Passive Redundancy, (4) Spare
	Recovery Re-introduction	(1) Shadow, (2) State Resynchronization, (3) Rollback
	Fault Prevention	(1) Removal from service, (2) Transactions, (3) Process Monitor
Modifiability	Localize Change	(1) Semantic coherence, (2) Anticipate expected change, (3) Generalize module, (4) Limit possible options, (5) Abstract common services
	Prevention of Ripple Effect	(1) Hide information, (2) Maintain existing interface, (3) Restrict communication paths, (4) Use an intermediary
	Defer Binding Time	(1) Runtime registration, (2) Configuration files, (3) Polymorphism, (4) Component replacement, (5) Adherence to defined protocol
Performance	Resource Demand	(1) Increase computation efficiency, (2) Reduce computational overhead, (3) Manage event rate, (4) Control frequency of sampling
	Resource Management	(1) Introduce concurrency, (2) Maintain multiple copies, (3) Increase available resources
	Resource Arbitration	(1) Scheduling policy
Security	Resisting Attacks	(1) Authenticate users, (2) Authorize users, (3) Maintain data confidentiality, (4) Maintain integrity, (5) Limit exposure, (6) Limit access
	Detecting Attacks	(1) Intrusion detection
	Recovering from Attacks	A. Restoration -- similar to Availability attribute
		B. Identification -- Audit trail
Testability	Manage Input/Output	(1) Record/Playback, (2) Separate interface from implementation, (3) Specialized access routines/interfaces
	Internal Monitoring	Built-in monitors
Usability	Separate User Interface	
	Support User Initiative	(1) Cancel, (2) Undo (3) Aggregate
	Support System Initiative	(1) User model, (2) System model , (3) Task model

To illustrate how a certain attribute can be achieved, consider the security considerations of the UMMCS in Figure 6. The UMMCS should restrict access to intruders and protect the system itself from virus attacks. Figure 6 illustrates the tactics embedded in the system design. Several methods that could be used to *authenticate users* are shown. A legitimate user can then access network data, and utilizes computing applications and services. A workstation could communicate within the network system via *encryption* in order to maintain *data confidentiality*. Our intended system would create an *intrusion detection system* in order to protect itself from attacks. *Firewall* within workstation is necessary to protect itself from intruders. Firewalls restrict access based on message source or destination port because messages from unknown sources may be a form of an attack. Finally, each server would maintain a *redundant copy of network data* in order for the system to maintain *data integrity*. Maintaining *audit trail* is necessary in order to track down hackers and attackers and to bring back the system to recovery.

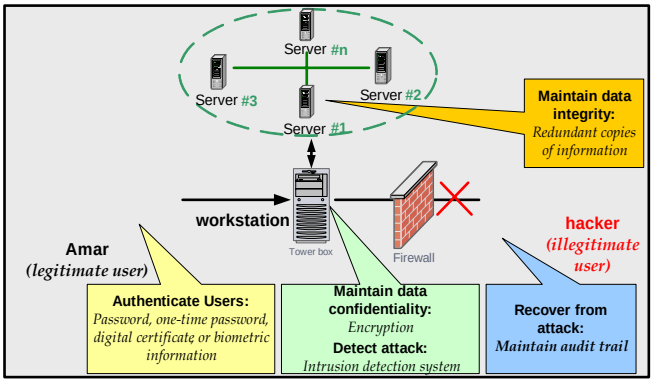

Figure 6. Applying security tactics in UMMCS.

Figure 7 demonstrates the ADD-decomposition of the Context Manager Agent. In this diagram, the availability attribute in the design is visible.

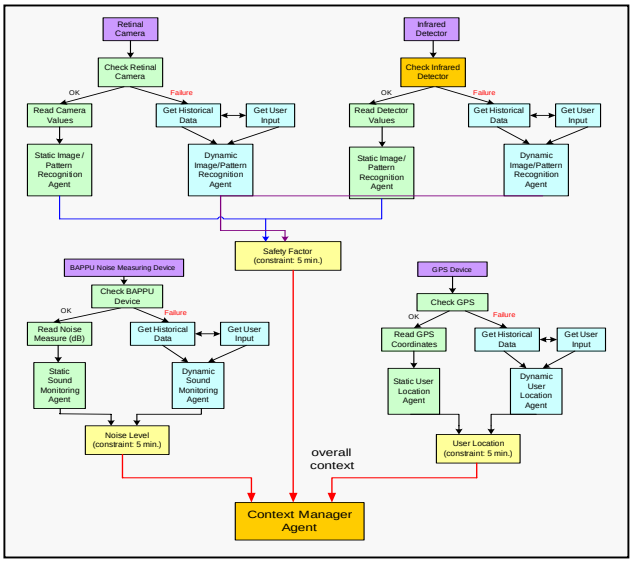

Figure 7. First level ADD-decomposition of CMA.

For example, in the event that any of the context sensors fail, the dynamic context agent would be invoked. To prevent the system from stalling or crashing due to sensor failure (resulting in an undeterminable overall context), the most relevant historical data recorded by the sensor would be used and the user is informed

about it. If the sensor is just fine, it would be read as usual to obtain the context measure. In the end, all these context agents would have reasonable results for the CMA to determine the overall context. Also, note that performance constraints is imposed (i.e. 5-minute time constraint) for the context agent to produce the necessary context assessment.

Again, due to space limitation, we could not provide the decomposition of the entire ubiquitous MM computing system. However, the concept that is applied on Figure 6 is the same one that should be applied in the decomposition of all system components.

4.2 Architectural Views

In concept, a view is a representation of a coherent set of architectural elements. It consists of a representation of a set of elements or components and the relationship among them. In some literature, architectural view and structure are sometimes used interchangeably. For an architectural design to demonstrate the necessary information of the interested stakeholders, one architectural view is often not enough; there needs to be two or three (or more) so that the management, analyst, programmers, end user and customer could see, understand and appreciate the architectural design based on each one's perspective.
In general, architectural views can come in three types, depending on the broad nature of the elements that they show:

1. Module. The elements are generally the modules, which are the units of implementation. Modules are code-based way of considering the system. This view shows the relationship among different modules.
2. Component-and-Connector. The elements are generally the runtime components (i.e. units of computations) and connectors (i.e. communication vehicle or protocol between elements). It satisfies the questions related to some shared data stores, parts of the system that are replicated, and parts of the system that run in parallel.
3. Allocation. This structure or view shows the relationship between software elements and the hardware or files that is created, used or executed.

Figure 8 illustrates the first-level modular view of the UMMCS. In the diagram, we have divided the complete system into three major architectural components: the TMA, the CMA, and the HKA. Each of these components is further divided into some sub-components. The relationship for this 1st-level decomposition is the lower module is a sub-module of the upper module. In general, the designer would create several decompositions until all modular elements are enumerated.

Figure 9 shows the first-level component-and-connector view of the UMMCS. The software structure shown in the diagram is far more complex than the one shown in the simple modular view of Figure 8. In Figure 9, a combination of client-server structure, process structure, concurrency and shared data structures are shown. For example, in the CMA, we could see that the processes "user location detection", "sound monitoring process" and "safety factor detection process" are all running concurrently. This is because each of these context activities must be executed in parallel, and their outputs are sent as inputs to the CMA in order that it can detect the user context correctly.

In general, the component-and-connector view is useful for performance analysis and load balancing for the client-and-server software structure. For process structure, the view is useful for performance analysis. For concurrency structure, the view is useful in identifying where resource contention could exist, and where thread may fork, join, be created or be killed. Shared data structure is useful for data integrity analysis.

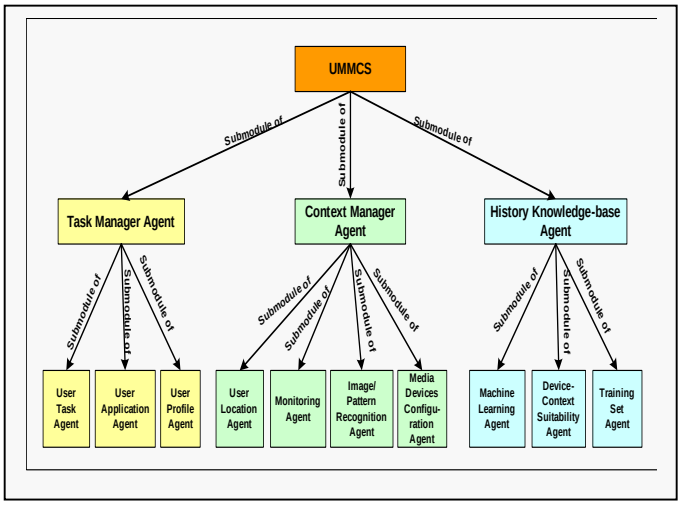

Figure 8. First-level Modular view of the UMMCS.

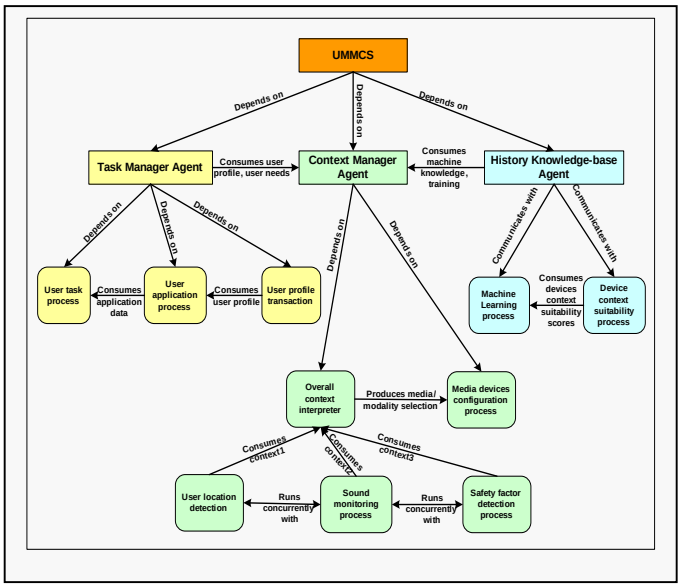

Figure 9. First-level component-and-connector view of the UMMCS.

Figure 10 demonstrates the allocation view of our system. We could see right away the resources that are involved in the system and how they are allocated to the running processes. For example, we can see that the global positioning system (GPS) is allocated to the user location processing; the BAPPU device is allocated to the sound monitoring processing, the camera with retinal recognition and the infrared detector are all allocated to the safety

factor detection processing, and the media devices configuration processing would migrate to the selected media and modalities because of their dynamic (rather than static) allocation relationship.

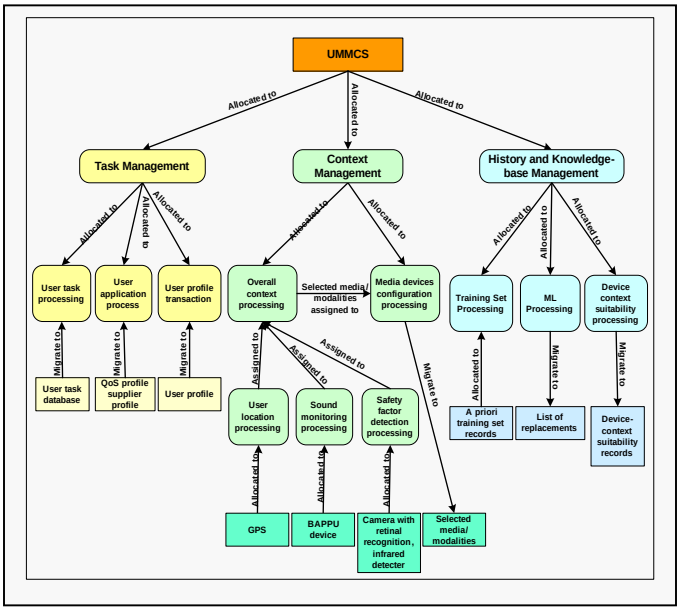

Figure 10. First level allocation view of the UMMCS.

5. ARCHITECTURAL EVALUATION

The Architectural Tradeoff Analysis Method (ATAM) [22] is the technique we adapt in evaluating our proposed architecture. In concept, our system architecture is still in the development stage hence this evaluation is apt only for this level of development.

ATAM is used to analyze system architecture. Although ATAM is used by different stakeholders with business driver motivations that we do not possess since our work is purely academic in nature, still ATAM is a good method to evaluate system architecture with respect to the prospect of achieving desired quality attributes. In software architecture, various styles or patterns exist, among them are: (i) *pipes and filters*, (ii) *data abstraction/object-oriented*, (iii) *event-based, implicit invocation*, (iv) l*ayered,* and (v) *repositories*. The choice of architecture style directly affects the quality attribute to achieve. For example, a layered pattern brings portability but at the expense of performance. A repository pattern is ideal on the producer-consumer type of system.

In general, the ATAM analysis involves the following steps: (1) *Presentation of ATAM*, (2) *Presentation of architecture*, (3) *Identification of architectural approaches*, (4) *Generation of quality attribute utility tree*, (5) *Analysis of architectural approaches*, (6) *Brainstorming and prioritizing scenarios*, and (7) *Presentation of results*. Figure 11 illustrates the layered architecture of the UMMCS. The layered architecture offers the perception of stimuli that could affect the achievement of desired system qualities.

On the hardware side, performance is again an issue. The devices and the sensors used to detect user context must be reliable. The fact that 1 context sample is generated in every 1 minute means that there is no room for devices and sensors to produce the needed data in a slower manner. The result of the ATAM analysis (i.e. *utility tree*) is shown in tabular form in Table 3. The quality attribute and the scenarios affecting the quality are described. The priority of importance in addressing these scenarios are denoted as *H* (if it is of high importance), *M* (for medium importance), and *L* (for low importance). For now, our priority is focused on the achievement of availability, performance, modifiability and scalability attributes. All other issues are of low priority.

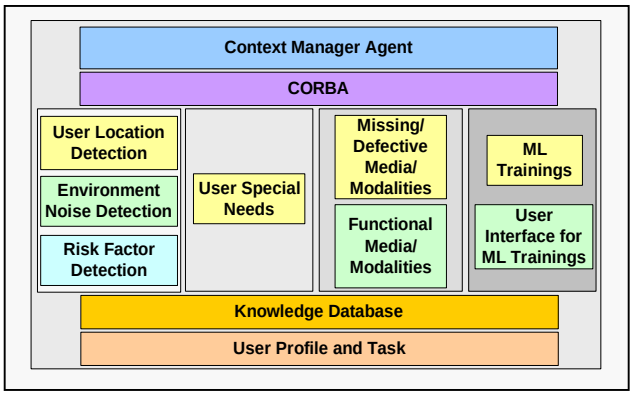

Figure 11. Layered view of the UMMCS.

Table 3. The tabular form of the utility tree for the UMMCS.

Quality Attribute	Attribute Refinement	Scenarios
Modifiability	Ease of modification	A user updates his user profile, inputting his preferences. The update and modification should be as simple and uncomplicated as possible. (H,M)
	Ease of update	The system could restart re-building the training set (latest version). This happens when new devices are introduced and are found to have a higher ranking than the old media and modalities. (H, M)
	Ease of modifying the incrémental ML training sets	The system should allow the modification of user entry (in case of error in data entry) in the selection of alternatives to missing/failed components. (H,M).
	Ease of adding extra parameters in detecting user's context	This is self-configuration with ease. When a new component or variable is introduced as part of detecting user context, adaptation of affected components should be as seamless as possible. (H, M).
Availability	Data consistency and integrity	When the user moves from one place to another, his user profile and user task data should be following him. This data should be the latest copies. (H,M)
	Replacement for faulty media or modalities	A media or modality of highest priority is missing or defective, the availability of replacement is vital for the post-condition to be preserved. In the event of multiple failures of devices, the availability of incrémental ML component should be as quick as possible (say 2 seconds). (H, M)
Performance	Detection of User context	The design of CMA should include a feature that automatically detects user context in not more than 6 minutes. The media/modalities that fit his context should be activated in such short time. (H, M).
Scalability	Increasing size of knowledge database	The file system should scale to the size of the knowledge database, which continuously grow. This is an issue for PDA. (H,M).
Modularity	Functional subsets	Building software components that could alternate from failed software components. (L,L)

6. CONCLUSION

In this paper, we have demonstrated in part the requirements analysis, the architectural design and evaluation of a ubiquitous

MM computing system. This system detects the user context and user needs in the user profile, and yields the media and modalities that are appropriate for the given user situation. The user context is based on user's location, the noise level in the workplace, and the presence or absence of other people in the workplace. It is also based on the computing device the user is utilizing. The special need, if applicable, is whether or not a user is handicapped. And if so, which handicap then. The requirements analysis is presented from the system's functional needs' perspective. The steps in software architecture design are presented based on the system's quality attribute perspective. The quality attributes and the stimuli that could challenge the system from achieving the desired system qualities are shown, although not presented entirely due to space restrictions. The quality attribute scenarios (QAS) are presented in general, and our system is considered as a test case. Our ongoing research includes system or system's component self-management amidst the growing complexity of information technology. Further works on autonomic computing qualities (i.e. self-healing, self protection, self-configuration, and self-optimization) could be injected into our system that is still in its development stage. We did subject our system to ATAM architectural evaluation, and true enough, as system architects, we could imagine and consider just too many scenarios, big or small, urgent or not, that could affect the system's capacity of achieving basic quality attributes. Our future works include the dynamic architecture and the system self-adaptation based on varying conditions such as resource restrictions (i.e. limited bandwidth) and varying user priorities.

7. ACKNOWLEDGMENTS

This work has been made possible by the funding awarded by the Natural Sciences and Engineering Research Council of Canada (NSERC)

8. REFERENCES

[1] H. Djenidi, S. Benarif, A. Ramdane-Cherif, C. Tadj, and N. Levy, "*Generic Multimedia Multimodal Agents Paradigms and their Dynamic Reconfiguration at the Architectural Level*," in *Eurasip Journal on Applied Signal Processing*, Hindawi Publishing Corp., USA, pp. 1688-1707, 2004

[2] M. D. Hina, A. Ramdane-Cherif, and C. Tadj, "*A Context-Sensitive Incremental Learning Paradigm of a Ubiquitous Multimodal Multimedia Computing System*," presented at WiMob 2005, IEEE International Conference on Wireless and Mobile Computing, Networking and Communications, Montreal, Canada, 2005.

[3] M. Satyanarayanan, "*Pervasive Computing: Vision and Challenges*," IEEE Personal Communications, vol. 8, pp. 10-17, 2001.

[4] M. Satyanarayanan, "*Mobile Information Access*," IEEE Personal Communications, vol. 3, pp. 26-33, 1996.

[5] M. Satyanarayanan, "*The Evolution of Coda,*" ACM Transactions on Computer Systems, vol. 20, pp. 85-124, 2002.

[6] K. Pahlavan and P. Krishnamurthy, "*Principles of Wireless Networks*," 2002.

[7] L. Bass, P. Clements, and R. Kazman, "*Software Architecture in Practice*," 2nd ed, 2003.

[8] D. Garlan and D. Perry, "*Software Architecture: Practice, Potential, and Pitfalls*," Sorrento, Italy, 1994.

[9] P. Clements, R. Kazman, and M. Klein, "*Evaluating Software Architecture,*" 2002.

[10] F. Bachmann, L. Bass, M. Klein, and C. Shelton, "*Designing Software Architectures to Achieve Quality Attribute Requirements*," IEE Proceedings: Software, vol. 152, pp. 153-165, 2005.

[11] J. Bosch and L. Lundberg, "*Software Architecture - Engineering Quality Attributes,*" Journal of Systems and Software, vol. 66, pp. 183-186, 2003.

[12] C. Stoermer, L. O'Brien, and C. Verhoef, "*Moving Towards Quality Attribute Driven Software Architecture Reconstruction*," Victoria, BC, Canada, 2003.

[13] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, "*Architectural Styles, Design Patterns, and Objects*," IEEE Software, vol. 14, pp. 43-52, 1997.

[14]T. Mitchell, "*Machine Learning*," McGraw-Hill, 1997.

[15] O. Buffet, A. Dutech, and F. Charpillet, "*Incremental Reinforcement Learning for Designing Multi-agent Systems,*" Montreal, Que., Canada, 2001.

[16] A. Caglayan, M. Snorrason, J. Jacoby, J. Mazzu, R. Jones, and C. River, "*Learn Sesame – A Learning Agent Engine*," presented at Applied Artificial Intelligence, 1997.

[17] S.-W. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, and P. Steenkiste, "*Using Architectural Style as a Basis for System Self-Repair*," presented at IFIP 17th World Computer Congress, 3rd Working IEEE/IFIP Conference on Software Architecture (WICSA3), Montreal, Quebec, Canada, 2002.

[18] M. D. Hina, A. Ramdane-Cherif, and C. Tadj, "*A Ubiquitous Context-sensitive Multimodal Multimedia Computing and Its Machine-Learning Assisted Reconfiguration at the Architectural Level*," presented at Workshop on Multimedia Information Processing and Retrieval, Proceedings of the Seventh IEEE International Symposium on Multimedia, California, USA, 2005.

[19] M. D. Hina, C. Tadj, and A. Ramdane-Cherif, "*Design of an Incremental Learning Component of a Ubiquitous Multimodal Multimedia Computing System*," submitted at WiMob 2006, 2nd IEEE International Conference on Wireless and Mobile Computing, Networking and Communications, Montreal, Quebec, Canada, 2006.

[20] F. Bachmann and L. Bass, "Introduction to the Attribute Driven Design Method," presented at Proceedings - IEEE Computer Society International Conference on Software Engineering, Toronto, Ontario, Canada, 2001.

[21] M. D. Hina, C. Tadj, and T. Ramdane-Cherif, "*Quality Attributes and Self-Management Considerations in Designing a Pervasive Multimodal Multimedia Computing System,*" presented at GPC 2006, The First International Conference on Grid and Pervasive Computing, Tunghai University, Taichung, Taiwan, 2006.

[22] R. Kazman, M. Klein, and P. Clements, "*ATAM: Method for Architecture Evaluation*," CMU/SEI 2000.

Open Source Software in Software Engineering Education: No Free Lunch

Pankaj Kamthan
Department of Computer Science and
Software Engineering
Concordia University, Montreal,
Quebec, Canada H3G 1M8
1-(514)-848-2424-3000

kamthan@cse.concordia.ca

ABSTRACT
The Open Source Software (OSS) movement has introduced a new way of developing and disseminating software. In this paper, we examine some of the fundamental practices in traditional software engineering education (SEE) from an OSS perspective. A simple framework as a first step for introducing OSS in SEE is presented. The opportunities and obstacles of OSS in SEE are identified and analyzed with the help of examples.

Categories and Subject Descriptors
D.2 [**Software Engineering**]; K.3.2 [**Computer and Information Science Education**]

General Terms
Management, Documentation, Economics, Experimentation, Human Factors.

Keywords
Open Source, Software Process, Software Engineering, Education, Quality.

1. INTRODUCTION AND BACKGROUND
The steady rise of Open Source Software (OSS) [14] over the last few decades has made a noticeable impact on many sectors of society where software has a role to play. As reflected from the frequency of media articles, traffic on mailing lists, and growing research literature, OSS has garnered much support in the software community. Indeed, from the early days of GNU software, to Linux and its utilities, to more recent the Apache Software Project, to name a few, OSS has changed the way software is developed and used.

The concept of open source can mean different things in different contexts [4]. For the purposes of this paper, we will use "open source" as a single encompassing term for "free/freely available" or "libre/liberated" software whose source is available without cost to the user, imposes minimal non-restrictive licensing conditions, and is itself based upon non-proprietary technologies. Software that does not fall into this category is termed as non-OSS. Commercial software is one class of non-OSS.

As OSS becomes prominent, the issue of its outreach in an educational context arises. This paper takes the position that students studying software development should be exposed to this rapidly growing area. In fact, the use of OSS in computer science education has been emphasized in recent years [1,7,8,12]. It has also been suggested [2] that developing OSS could also help students in their future career paths.

However, the current studies of OSS-based education are limited in one or more of the following ways: the discussion is often confined to the case study of a specific OSS, coverage tends to be one-sided with general conclusions, do not highlight the problems associated with introducing OSS, do not address SEE exclusively, or ignore aspects of software engineering that OSS do not address. One of the purposes of this paper is to address these concerns.

The rest of the paper is organized as follows. In Section 2, selected SEE practices are addressed in the light of OSS. Section 3 presents an elementary framework for introducing OSS in SEE. Finally, in Section 4, we conclude with challenges and directions for future research.

2. SEE AND OSS: SIMILARITIES AND DIFFERENCES
Before embarking on an OSS-based development, we need to inspect how it manifests itself in a traditional SEE setting. This section looks at six broadly classified aspects, namely that of management, process, modeling/specification, standards, documentation, and quality, that are common in most SEE contexts and examines how they are realized (or not) in an OSS environment.

2.1 Management
We shall limit our discussion to measuring success and team, time, and configuration management.

The goals of developing software in educational and OOS contexts are different. In SEE, the software product is a means to the end, not an end in itself. It has been reported [2] that OSS often lacks precise specification of goals and as a result fails to define "success". The reason for abandoning an OSS project are often not given or made public. In SEE, there is a price for not performing up to the expectations or working to your full potential, and is exhibited in a grading differential.

There are differences between the social structure of a team of students in an SEE environment versus participants in the OSS development. In general, course project teams in SEE are collocated while those in OSS are distributed. There is also a notable difference with respect to social bonding. The students most likely belong to the same institution, may take multiple

courses together. The students also may be related on a personal level (roommates, siblings, friends), while that is not the norm in an OSS development where the participants are loosely related. There is no inherently hierarchical team structure in OSS. Since participation does not require qualification and is voluntary, anybody can participate, and at times, anybody does. It is well-known in software project management studies that it is not the number that matters. There is usually a core group that contributes the most with a sporadic participation by others [9]. On the other hand, assuming responsibility and accountability are at the heart of SEE. Students are (or learn to be) accountable to others in their team as well to the teacher.

In lieu of mimicking real-world software projects as well as due to natural limitations of schedules at educational institutions, there are inevitable time constraints associated with course projects. However, there is little sense of urgency in OSS projects.

The distributed nature of contribution by anybody at any time as well as the desire of the developers to be able to disseminate "up-to-the minute" code has led to a usually strong support for configuration management (version control, bug tracking, or build management) in OSS development. Posting nightly builds for tryout is quite common in an OSS environment. However, in author's experience with SEE, configuration management is not as pervasive as OSS and is usually limited to version control and backups.

2.2 Process

In SEE, students are normally introduced to both rigid/prescriptive and flexible/agile process models. OSS development process, known as the "Bazaar model" [13], is not subsumed by any of these although it is much closer to the latter than it is to the former.

As an example, many of the practices of Extreme Programming (XP) are applicable to OSS [11]. However, two of the key practices of XP, namely that of Onsite Customer and Pair Programming, do not scale well to OSS.

Unlike the case of traditional software process environments where organizations make the use of Capability Maturity Model (CMM), there is little systematic effort towards maturity of OSS process.

2.3 Modeling/Specification

Modeling, particularly during early phases of software development, is playing an increasingly important role in activities and deliverables in SEE. It is often emphasized that early modeling is crucial from the point of view of understanding the problem and solution domains in an implementation neutral manner, and control and prevention of problems that can propagate into later stages.

The Unified Modeling Language (UML) has emerged as a standard language for modeling the structure and behavior of object-oriented systems, and its use in the last few years in SEE has increased dramatically. However, there is little evidence of use of UML, and in general of any form of systematic modeling, in OSS.

Formal Specifications are also integral to many courses in SEE where the safety requirements or design of a critical system need to be precisely (mathematically) expressed. However, once again, there is little evidence to support the use of mathematics in OSS problem or solution domains for system analysis or synthesis, respectively. This evidently limits the use of OSS, even in part, in safety-critical software.

2.4 Standards

There are a variety of reasons for introducing and adhering to standards in SEE. Standards provide a common ground for a team, streamline efforts, and when applied well are known to contribute towards quality improvement [14].The author is a strong proponent of the use of standards throughout SEE and has made mandatory use of IEEE and/or ISO/IEC standards in process documents and strongly encouraged standardized (ANSI, ECMA) definitions of programming languages and corresponding compilers/interpreters.

The OSS approach serves as a platform for trying out new technologies and developing "proof-of-concept" implementations, and, in doing so, the use of standards is limited to data formats such as the HyperText Markup Language (HTML) or the Extensible Markup Language (XML).

2.5 Documentation

The role of documentation is usually accentuated in SEE. The courses related to technical communication and programming methodology early in the curriculum form the basis of internal documentation of software developed in later courses. In some cases, creating external documentation (user manual) may also be required.

In contrast, it has been the author's experience that OSS is in general weak with respect to documentation. The documentation at times may not be complete or may only be sketchy. The OSS style of writing also at times tends to be 'wordy', and 'casual' rather than succinct and technically-inclined to the issue at hand. At times, help or tutorial documents are not updated to synchronize with the latest code releases.

2.6 Quality

In SEE, there is much emphasis on quality and its relation to measurement in all aspects of software (project, process, product, and occasionally even people).

There are many OSS that exhibit high quality. However, the approach to quality assurance and assessment is not systematic and therefore seemingly not repeatable. In OSS, peer reviews are used as a technique for an informal evaluation whereas formal inspections are apparently non-existent. Comprehensive collections of test cases, test suites or test harnesses are rare, and broad testing is even rarer. More importantly, participation is voluntary and monitoring is almost non-existent. The linear relation of number of bugs found to improvement of quality proposed by the OSS development process [13] is overly simplistic, and has indeed been termed as a "fallacy" from a software engineering perspective [6]. The issue of OSS quality in general, and concerns of security and usability in particular, has been addressed in [9,10].

Having analyzed the parity/disparity between OSS and traditional SEE, we now turn our attention to realizing OSS in SEE.

3. INTRODUCING OSS IN SEE

The OSS ecosystem can be applied in a SEE context in a number of different (but not necessarily mutually exclusive) ways: OSS for pedagogy and for learning, as a CASE tool, as a sub-system, for reuse, and adoption of the OSS process.

A project without clear goals will not achieve its goals clearly [5]. Therefore, the aforementioned ways need to be aligned with teaching and learning goals. Furthermore, since software engineering is a practical discipline, all the aims and activities from their initiation to their completion should be feasible. Finally, use and/or development of OSS must be legally acceptable in the place where SEE is carried out. Table 1 illustrates the architecture of the framework.

Table 1. A high-level view of the OSS/SEE framework

<table>
<tr><td rowspan="2">Theory</td><td>OSS for Pedagogy</td><td rowspan="6">Teaching and Learning Goals</td><td rowspan="6">Feasibility</td></tr>
<tr><td>OSS for Learning</td></tr>
<tr><td rowspan="4">Application</td><td>OSS as CASE Tool</td></tr>
<tr><td>OSS as Sub-System</td></tr>
<tr><td>OSS for Reuse</td></tr>
<tr><td>OSS Process Adoption</td></tr>
<tr><td colspan="4">Legality</td></tr>
</table>

The precise articulation of the teaching and learning goals, of the criteria and techniques to be adopted for carrying out a feasibility study, or of legal issues, is beyond the scope of this paper. We simply state that in an educational environment, both the teachers and students are bound by time constraints. Moreover, educational institutions and students are increasingly facing budgetary constraints. Any efforts of integrating OSS in SEE could be potentially threatened if either of these is severely violated. It would also undermine one of the founding philosophies of the OSS, that is, of cost associated with software. Any feasibility analysis ultimately requires making *decisions* to prioritize among the given options. To help achieve that, Analytical Hierarchy Process (AHP) and Quality Function Deployment (QFD) are two commonly used project management techniques. Any feasibility analysis, however, should also be in agreement with the institutional emphasis on decision support for software engineering in general.

Any SEE must occur within the legal framework of the country where it is carried out. We note that laws can be locality-dependent and thereby present obstacles to use of OSS in SEE. For example, the freedom of deploying OSS in Canada does not currently carry over to Germany. OSS related policies may also vary across provinces and educational institutions.

We now discuss the different uses of OSS in SEE.

3.1 OSS for Pedagogy

This approach to OSS in SEE advocates use of OSS for teaching purposes in a classroom. Teachers could also include OSS as part of the course content, something that is unique to OSS environment and not readily possible in a non-OSS context. The availability of source code in OSS provides a unique opportunity for the teacher to experiment. Source code internals of software (that is usually larger in scale than those accompanying the commonly used textbooks) can be shown and its quality can be debated. Teachers could, for example, point out both successful and failed OSS efforts, and reasons for being so.

OSS could also be used by students in making classroom demonstrations and presentations. For example, the use of slides in Extensible HTML (XHTML), projection media presentation semantics supplied by a style sheet in Cascading Style Sheets (CSS), and used in conjunction with the Amaya user agent provides a simple, non-binary, cost-effective, interoperable, and enduring alternative to Microsoft PowerPoint.

3.2 OSS for Learning

This approach to OSS in SEE advocates use of OSS for self-learning purposes outside classroom (say, at home). The ascent of affordable personal computers, ubiquitous and high-speed Internet connectivity, and the use of the Web as an information base is having a major impact on the way students study and learn at home. In contrast with non-OSS, the availability of OSS source code provides a unique opportunity for the student to experiment. Thus OSS becomes a platform for skills development. One of the constructivist theories of learning [16] has emphasized learning by doing.

There are, however, certain hindrances when this is put into practice. As pointed out in Sections 2.5 and 2.6, lack of sufficient documentation or quality could also pose obstacles to novice. Technical support mainly for special-purpose OSS is usually limited to participation in an electronic forum with no guarantees of a timely response, if at all.

3.3 OSS as CASE Tool

This approach to OSS in SEE advocates use of OSS for supporting development activities (that is, as CASE tools). We need software to develop software, and OSS utilities could prove useful in that regard. Examples are Apache Maven for project management, Yahoo! Groups for fostering team-wide communication, ArgoUML as a UML modeler, GNU Emacs as a universal text-based editor or alternatively IBM Eclipse as an multi-purpose authoring environment, CCDoc for C++ documentation, Bugzilla for issue tracking, Apache Ant for building, Lint for code styling, JUnit for unit testing, and Subversion for placing deliverables under version control, to name a few.

However, OSS does not always scale well in comparison to their non-OSS counterparts. For example, in spite of their relatively high cost, IBM Rational Rose or Borland Together ControlCenter still remain the UML modelers of choice. In some cases, unless required otherwise, students may also find “all-in-one” multi-utility packaged commercial integrated development environments (IDEs) more convenient to use for programming purposes.

3.4 OSS as Sub-System

This approach to OSS in SEE advocates use of a standalone OSS as auxiliary software that supports the system under development

for the course project. In that regard, OSS support has in general been exemplary.

For example, a project involving a Web Application could use Amaya as the user agent on the client-side and Jigsaw or Apache Web Server along with MySQL/PHP and (one of the many available) XML parser on the server-side.

This also highlights one of the major points of departure between OSS and non-OSS (which would in general not allow copy and redistribution outside the realm of the customer, and even that with strict restrictions).

3.5 OSS for Reuse

This approach to OSS in SEE advocates reuse portions of OSS code in assignments or as part of the system under development as for the course project. Examples include OOS libraries or frameworks. It ameliorates the tedium of writing the entire code from scratch, particularly that for routine primitive functions such as finding the LU decomposition of a matrix or drawing an elliptic hyperboloid.

However, issues of the students treating reused code as a "black box" without really understanding the internals, degree to which reuse should be allowed, and that of appropriate acknowledgement remain a challenge. We also note here that according to the COCOMO II cost estimation model, reuse comes at a price of learning and adapting to new situations.

The following example illustrates some evaluation issues related to reuse. Suppose program A (reuse) submitted by one student and program B (no reuse) submitted by another student as part of their work are being evaluated for an external quality characteristic, say performance, by the teacher. Now, should A be graded higher than B if the teacher determines that it is the reused code that is making the difference? Should the "art" of finding and reusing OSS matter? These questions need to be addressed and answered to make OSS reuse viable.

3.6 OSS Process Adoption

This approach to OSS in SEE advocates the adoption of OSS practices and develop software as part of a course project. The resulting software will then itself be an OSS and whose development will be open to public. As an example, SourceForge could provide a medium for development, collaboration, and distribution.

However, this may be the most challenging of all the dimensions. The Bazaar model requires a different mindset from traditional approaches and may need to be "tailored" for an educational use. For example, instilling the sense of team work in physical proximity, and experiencing the issues that go with it are an important part of learning. Reports of a successful application are also rare [1].

Also, fairness in evaluation is still an issue. For example, should (un)solicited feedback from those not registered in the course be allowed? If so, how should a feedback imbalance across teams be dealt with? Again, these questions need to be addressed and satisfactorily answered prior to any OSS initiative.

4. CONCLUSION AND FUTURE WORK

Today, OSS has reached the level of maturity that it could be embraced as well as criticized, but not ignored. If the predictions of software business models [2,3] are correct, OSS and non-OSS will continue to co-exist. Both OSS and non-OSS have their own share of strengths and weaknesses, are most likely to co-exist, and any approach to SEE should take them into consideration.

If one of the goals of SEE is to prepare students for their future careers, we must look at the OSS objectively. For that the SEE culture in educational institutions will need to evolve.

OSS has much to offer to SEE. However, the transition from one to the other is hardly straightforward. Any adoption of OSS in SEE needs to be aware of the philosophical differences between the two and prepare accordingly. The adoption of OSS in SEE need not be seen with skepticism but rather with cautious optimism.

A few directions of research emanate from this work. Among the possible domains that OSS addresses, it would be of interest to examine the ones more congruent to SEE. Among the open source possibilities, this paper focuses mostly on OSS; a natural extension of this work would be to look into the use of "open content" (excluding source code) in SEE. The aim of open content is to "facilitate the prolific creation of freely available, high-quality, well-maintained content" (not including software). MIT OpenCourseWare and Rice Connexions are two commonly cited examples of institution-initiated efforts of making course content open to public-at-large. The significance of open content for education in general has been highlighted in [1]. The continually increasing price of textbooks, none of which may be suitable as-is to a given course, is one motivation open content in SEE. We plan to investigate these in future work.

5. ACKNOWLEDGMENTS

This work has benefited from discussions with software engineering students at Concordia University, Canada. I would especially like to thank Hsueh-Ieng Pai for careful reading, comments, and feedback.

6. REFERENCES

[1] Attwell, G. What is the Significance of Open Source Software for the Education and Training Community? *The First International Conference on Open Source Systems (OSS 2005), Genova, Italy, July 11-15, 2005*.

[2] Cusumano, M. A. Reflections on Free and Open Software. *Communications of the ACM*, 47, 10 (2004), 25-27.

[3] Feller, J., Fitzgerald, B., Hissam, S. A., and Lakhani, K. R. *Perspectives on Free and Open Source Software*. MIT Press (2005).

[4] Gacek, C. and Arief, B. The Many Meanings of Open Source. *IEEE Software*, 21, 1, IEEE Press (2004), 34-40.

[5] Gilb, T. *Principles of Software Engineering Management*. Addison-Wesley (1988).

[6] Glass, R.L. *Facts and Fallacies of Software Engineering*. Addison-Wesley (2003).

[7] González-Barahona, J. M., Heras-Quirós, P. D. L., Centeno-González, J., Matellán-Olivera, and Ballesteros-Cámara, F. Libre Software for Computer Science Classes. *IEEE Software*, 17, 3 (2000), 76-79.

[8] Liu, C. Adopting Open-Source Software Engineering in Computer Science Education. *The Third Workshop on Open Source Software Engineering, Portland, USA, May 3, 2003*.

[9] Michlmayr, M., Hunt, F., and Probert, D. R. Quality Practices and Problems in Free Software Projects. *The First International Conference on Open Source Systems (OSS 2005), Genova, Italy, July 11-15, 2005*.

[10] Nichols, D. M. and Twidale, M. B. The Usability of Open Source Software. *First Monday*, 8, 1 (2003).

[11] Nishinaka, Y. Open Source Software Developments in XP Style. *The First Workshop on Open Source Software Engineering, Toronto, Canada, May 15, 2001*.

[12] Port, D. and Kaiser, G. Introducing a "Street Fair" Open Source Practice within Project Based Software Engineering Courses. *The First Workshop on Open Source Software Engineering, Toronto, Canada, May 15, 2001*.

[13] Raymond, E. S. *The Cathedral & the Bazaar*. O'Reilly & Associates (1999).

[14] Schneidewind, N. F. and Fenton, N.E. Do Standards Improve Product Quality? *IEEE Software*, 13, 1, IEEE Press (1996), 22-24.

[15] Vixie, P. Software Engineering. In: *Open Sources: Voices from the Open Source Revolution*. C. DiBona, S. Ockman, M. Stone (Editors). O'Reilly & Associates (1999).

[16] Vygotsky, L. S. *Mind in Society: The Development of Higher Psychological Processes*. M. Cole, V. John-Steiner, S. Scribner, E. Souberman (Editors). Harvard University Press (1978).

How Valid is the Notion of "Information Society"?

Mohamed Ben Moussa
Department of Communication Studies
Concordia University, Montreal, Quebec
m_benmoussa@alcor.concordia.ca

ABSTRACT

The notion of "Information Society" entails that new Information and Computer Technologies and Software have radically transformed the industrialized countries, leading in the process to new political, social and cultural structures. Yet, the notion of the Information Society reflects a futuristic view that is based on a deterministic perception of technology and its impact on society.

General Terms

Global, impact, production, democracy, corporations, new economy, power.

Keywords

Information Society, knowledge, Computer hardware and software, ICT, Network, Informational economy.

1. INTRODUCTION

The impact of technology in general and software in particular are not limited to areas of industry and business. They have far reaching effects on all aspects of life, including social, political and cultural dimensions. Various concepts and theories have been used to understand, analyze and record this impact, the most important of which is the notion of 'information society'. In fact, there is a unanimous consent that information and communication technologies (ICTs) and software have become pervasive in all aspects of life; but whether this pervasiveness has radically transformed society and has led to the emergence of a new one remains a problematic question that still needs to be verified.

The information society is one where information technologies and software permeate all levels and sectors of society. It is also one which relies in its economic activities on knowledge and information as principal resources and products. And according to the advocates of this notion, the information society has already become a reality and replaced older forms of society.

However, its widespread attractiveness notwithstanding, the idea of the information society is far from convincing or conclusive. It is undeniable that information technologies and software are having a significant bearing on modern society and have even contributed effectively to transforming many of its aspects. This view, however, reflects a deterministic view that ignores that technology has a potential that can be an instrument of progress and conservatism, freedom and control, social inclusion as well as exclusion.

This paper will argue that the idea of the information society is not a valid proposition. It will begin by highlighting the main premises upon which this concept is built, drawing mainly upon the ideas of one of the most prominent theorists of the 'information society', namely Manuel Castells. This choice comes from the fact that Castells' is the most comprehensive theory of the information society as it covers many aspects from economy to culture and politics.

2. THE POLITICAL ECONOMY OF THE INFORMATION SOCIETY:

At the economic level, Castells maintains that the information society has developed due to the transformation of the industrial system of production and the emergence of the 'informational capitalism'. He (1996:17) points out to a number of features that characterize this new economy, including knowledge, which acts 'as the main source of productivity', and a high level of globalisation and networking, since it is an economy that works as a unit in real time on a planetary scale (ibid:101). Moreover, the informational economy knows a decline in manufacturing employment due to the 'rapid rise of managerial, professional and technical jobs' (ibid: 244-5).

Besides, Castells advances that power relationships between capitalist classes and 'knowledge workers' have radically changed. In this regard, he (1996: 104) asserts that 'ICTs have reduced the effectiveness of global corporations and dramatically empowered those people and organizations who are entrepreneurial and effective in terms of networking'. Because knowledge and information are the essential factors in production, 'the new producers of informational capitalism are […] knowledge generators and information processors' (Castells, 1998: 345).

As to the political level, Castells claims that the spread of the global information networks announces the demise of the nation state and the rise of new forms of politics. 'Bypassed by global networks of wealth, power, and information, the modern nation state has lost much of its sovereignty' (Castells: 1997, 354). Likewise, political parties and the whole civil society have been severely weakened since they 'find themselves deprived of actual meaning in the new social context' (ibid: 355). In fact, Castells believes that the rise of the information society has brought about a new form of politics based on ICTs that is providing marginal groups and international movements with effective means to further their causes (Castells, 1998: 68-135), and is leading to a more participatory form of democracy (2000: 391-392).

3. LIMITATIONS OF 'INFORMATION ECONOMY' MODEL:

There is no evidence that industrial economies have become more oriented towards information-based sectors at the expanse of the manufacturing sector. Stehr (2004: 216) maintains that the 'share of the manufacturing sector between 1978 and 1990 has declined somewhat in some of the countries, remained stable in others and increased in the case of the Japanese economy'. This testifies to the fact that the manufacturing sector has either largely conserved its position in the economy or has increased in importance. Besides, the fact that the world economy is operating as a network is not a new phenomenon. Garnham (2004: 173) rightly affirms that communication networks have underpinned the capitalist mode of production with extraordinary speed and reach since the 19th century. The new networks are, therefore, an extension of that system.

Similarly, there is no clear evidence that there is a radical shift towards information or knowledge-based labour. Many of the criteria on the basis of which information workers are singled out by the information theorists are questionable. Though Castells uses a variety of criteria to single out information workers, he does not specify what is the sufficient knowledge in ICTs that distinguishes knowledge workers. Webster (2002: 115) claims that 'the journalist on a daily newspaper is to Castells an informational worker in much the same way as is the surgeon in a hospital'. The question of criteria takes a problematic proportion for Stehr (2004: 218) who asserts that 'what is needed even more urgently, independent of conventional occupational labels, is a valid examination of actual work tasks carried out by employees '.

Equally important, the advance of the new ICTs has not undermined the grip of multinationals on economy. Indeed, the current drift towards mergers and takeovers among multinational corporations that control IT industries reveals that these companies are far from losing control on the market; rather, they are consolidating their grip on it. Ramonet (2003) argues that the development of new ICTs has opened the way for giant corporations to control three distinguished fields, namely 'mass culture with its commercial logic[..]; communications, as advertising, marketing and propaganda; and news and information, represented by agencies, radio and television news'. These technologies have, thus, allowed giant corporations to expand their business, to extend their control to different fields on a global scale and to break barriers between different domains as 'distinctions between office, home, work and leisure' have become insignificant (Kumar, 2004: 115). Consequently, what is called 'information society' is, in fact, merely an intensification and continuation of the capitalist system that has been developing over the whole twentieth century.

On the other hand, there is no indication that power relations between capitalist classes and 'knowledge labour' have changed. Intellectual property rights that govern most scientific research in the world are controlled basically by large corporations and firms. According to May (2002: 73), capitalists, rather than knowledge labour, control the means of production by patenting processes and technical procedures needed for knowledge work. Queau (2000) claims that firms and interest groups have been lobbying successfully to tighten intellectual property rights 'by using the 'multimedia revolution' as an argument'.

Besides, while Castells (1998: 346) celebrates the emergence of a new class of self-employed knowledge producers in full control of their work process, this mode of employment is, in reality, imposed in many cases by the corporations to rise benefits. As an example, Microsoft employs thousands of part-time employees and to avoid paying them the benefits to which full-time employees are entitled, it adopted a policy stipulating that 'those who have worked for the company for a year are required to take a thirty-one day break before being rehired as temps' (CCMS, 2003). In this regard, Garnham (2004:178) argues that 'the shift from energy to brainpower does not necessarily change the subordination of labour to capital'.

4. LIMITATIONS OF THE ELECTRONIC-DEMOCRACY MODEL:

In the political sphere, though the development of the new technologies challenges the power of the nation state, ICTs have been used to strengthen the hold of the former on society. Upon their invention, older forms of technology, such as the telegraph, the telephone and the radio, appeared to threaten the ability of states to control their territories and people. Yet, nation states managed to use them to further strengthen their power. Likewise, the new technologies provide them with unlimited means of surveillance and control over their citizens. Robins and Webster (2004: 72) maintain that 'technologies have increasingly been deployed in the twentieth century to render the exercise of power more efficient and automatic'. It is, therefore, erroneous to state that the rise of the information society is leading to the demise of the state.

Besides, it is highly improbable that the new ICTs and 'the network society' are causing the decline of the traditional political system. Though television has changed the way political campaigns and politics, in general, are communicated, it has not led to the loosening of the elites' grip over power. Instead, it has given them a powerful tool of propaganda and manipulation of the masses. As an illustration, in the United States, '78 percent of political web pages fell within the mainstream of American political culture', while 'the political users of the net in the United States are more inclined than the average citizens to vote for the major political parties' (Curran & Seaton, 2003: 265).

Equally important, the ability of the new ICTs to create a real participatory democracy is too optimistic and unrealistic. Contrary to Castells' claim about the advance of new democratic projects such as the Digital city (2000: 391-2), research studies on the role of ICT in promoting a participatory democracy have established that this role is very limited. An experiment of a political debate conducted online over one month and comprising different people revealed that most discussions were dominated by a limited number of people (Papacharissi, 2002: 5), usually the same people who are politically active in offline politics. Van Dijk (2000: 182) rightly maintains that the 'biggest mistake' usually made in that domain is to presume that technology can 'solve fundamental problems of citizen participation' while in reality this problem has 'deep social, cultural and mental roots'. Along the same lines, Lax (2000: 165) affirms that there is no evidence that cyber communities have any significant effect on politics in the real world since many of them are 'short-lived, with a flurry of electronic activity for a while before falling out of favour with participants'.

5. GENDERED TECHNOLOGY AND THE LIMITATIONS OF 'REVOLUTION' MODEL:

Morrison and Svennevig (2001: 127) argue that the concept of a revolution implies 'a radical change in social organization […] and a radical change in the way people lead their lives'. In this light, the claim that new cultural structures are reinventing a new order at the level of gender relations is exaggerated. In fact, although women are catching up with men in terms of access to the new ICTs, there are many indications that the type of use and appropriation of these technologies is still shaped by dominating cultural paradigms. In UK, for example, while the number of girl students in higher education exceeds that of the boys, the number of women studying IT courses is still very low. In fact, the number of women studying IT and computing has even fallen in recent years and represents only 18% of overall students (Trayhurn, 2002: 93). This fact reflects a cultural bias in society that considers IT and computing more suitable to boys than to girls, as boys start using computers from a young age (ibid: 96).

The differences between boys and girls as regards the use of the internet also reflects established gender order in society. A research study in UK revealed that while 50% of boys spend

their time on the net downloading music and software, most girls spend their time sending e-mails and using chat rooms (Kirkup, 2002: 50). And in Canada, it was found that while 'males are more interested in how technology works, women are more interested in the place it occupies in the wider social context' (UNESCO, 2002: 26). Thus, existing cultural structures that determine gender order in society are shaping the use of the new technologies. Kirkup (2001: 46) rightly points out that 'gender is having a stronger impact on the social and cultural production of technology' than the reverse.

6. REAL VS. VIRTUAL: THE PRODUCTION OF CULTURE IN THE 'NETWORK SOCIETY':

It is also equally difficult to believe that a radically new culture is emerging from the interaction between the networks and ICTs, on the one hand, and people and society, on the other. Castells' concept of 'real virtuality' proclaims a culture disembodied from the local experience and rooted in the timelessness of the computer networks (1998: 350). However, the networks and the new media themselves are deeply rooted in the time and place dimensions. Van Dijk (1999: 133) observes that 'nobody will deny the extreme relevance of (clock) time in the most advanced nerve-centres of ICT- the stock markets', while the place remains crucial for the new media as they will always depend on their 'material, social, physical and biological substructure or context' (ibid: 134).

The culture reflected by the new media is also rooted in the local experience and governed by the past, the present and the future. By surveying some 4000 websites, Halavais (2000) finds that the web conforms to traditional national borders. He points out that 'web sites are in most cases more likely to link to another site hosted in the same country than to cross national borders' (ibid: 7). In fact, virtual communities themselves are rooted in the social reality because 'without real people and real organizations, 'virtual communities' and 'textual cyberspace' would not exist' (Selvin, 2000: 7). Indeed, the production of culture, like the construction of identity, will continue to be affected by the experience rooted in real life within the dimension of real space and real time.

7. CONCLUSION

The notion of the information society is an attractive idea because it seems to account for the many transformations taking place in the world. However, it fails to demonstrate that these transformations constitute a total break with the past, rather than a continuation of it. Moreover, allocating a central role to technology and networks as the principal actors in society reflects an ideological stance that plays down the responsibility of politicians on their decisions and people's capacity to control their destinies. Information technology and software will continue to shape our lives, but the effect of technology itself is determined by the way people interpret it, put it into use, and weave it into the fabric of their social, political and cultural structures.

8. REFERENCES

Castells, M. (1996) *The Information Age: Economy, Society and Culture, Vol. 1: The Rise of the Network Society*, Massachusetts and Oxford: Blackwell.

Castells, M (1997) *The Information Age: Economy, Society and Culture, Vol. 2: The Power of Identity*, Massachusetts and Oxford: Blackwell.

Castells, M. (1998) *The Information Age: Economy, Society and Culture, Vol. 3: End of Millennium*, Massachusetts and Oxford: Blackwell.

Castells, M. (2000) *The Information Age: Economy, Society and Culture, Vol. 1: The Rise of the Network Society*, Oxford; Malden and Massachusetts: Blackwell Publishers.

Curran, J. and Seaton, J. (2003) *Power without Responsibility*, London and new York: Routledge.

Halavais, A. (2000) 'National borders on the world wide web', *New Media and Society*, Vol.2/1. London: Sage Publication.

Kirkup, (2001) ' Getting Our Hands On It: Gendered Inequality In Access to Information and Communication Technologies' in S. Lax (ed.) *Access Denied In the Information Age*, Basingtoke: Palgrave.

Kumar, K. (2004) "From Post-Industrial to Post-Modern Society", in F. Webster (ed.) *The Information Society Reader*, London: Routledge.

Lax, S. (2001) "Information, Education and Inequality: Is New Technology the Solution?", in S. Lax (ed.) *Access Denied In the Information Age*, Basingtoke: Palgrave.

May, C. (2002) *The Information Society: A Sceptical View*, Cambridge: Polity Press.

McQuail, D. (2000) *McQuail's Mass Communication Theory*, London, Thousand Oaks and New Delhi: Sage Publications.

Morrison, D.E. and Svennevig, M. (2001) "The Process of Change: An Empirical Examination of the Uptake and Impact of Technology", in S. Lax (ed.) *Access Denied In the Information Age*, Basingtoke: Palgrave.

Papacharissi, Z. (2002) 'The Virtual Sphere: the Internet as a public sphere', *New Media and Society*, Vol. 4/1. London: Sage.

Queau, P. (2000) 'Who Owns Knowledge?' in Le Monde Diplomatique, at http://mondediplo.com/2003/10/01media?var_recherche=Ramonet%2B+2003 (Retrieved on 12 February, 2004)

Ramonet, I. (2003)Ramonet 'Set the Media Free' in Le Monde Diplomatique, at http://mondediplo.com/2003/10/01media?var_recherche=Ramonet%2B+2003 (retrieved on 16 February, 2004)

Robins, K. and Webster, F. (2004) "The Long History of the Information Revolution", in F. Webster (ed.) *The Information Society Reader*, London: Routledge.

Stehr, N. (2004) "The Economic Structure of Knowledge Societies", in F. Webster (ed.) *The Information Society Reader*, London: Routledge.

Trayhurn, D. (2001) "Brickies or Bricoleurs? Gender in Computing and Design Courses", in S. Lax (ed.) *Access Denied In the Information Age*, Basingtoke: Palgrave.

UNESCO, (2002) ' Information, Communication and Knowledge: Building Contemporary Societies', at http://www.unesco.ca/english/Culture/Final_report_En/Pages%201%20to%2020%20from%20UNESCO_Report_Info_ENG2.pdf (Retrieved on 12 February, 2004)

Van Dijk, J. (1999) "The One-dimensional Network Society of Manual Castells", *New Media and Society*, Vol. 1/1, pp 127-138.

Van Dijk, J. and Hacker, K. (eds) (2000) *Digital Democracy: issues of theory and Practice*, Cambridge: Polity Press

Webster, F. (2002) *Theories of the Information Society*, London: Routledge