ICS 221 Topic/Paper summary for the tenth week

Submitted by: **Sushil Bajracharya** (sbajrach@ics.uci.edu)

Topic: Field Research in Software Engineering

Paper selected for the summary: [deSRD03] de Souza, C.R.B., Redmiles, D., Dourish, P., "Breaking the Code", Moving between Private and Public Work in Collaborative Software Development

Summary of the Paper

This paper documents the result of an empirical survey that discovered some important practices followed by the members of a collaborative software development team. Some of those practices were not formally identified as a part of the software development process and were pertaining to the activities and procedures developers took care of while they moved their work between *private* and *public* project workspace.

In a development environment where multiple stakeholders work on same set of documents concurrently, different kind of configuration management (CM) tools are employed that impose policies about the common and controlled usage of project artifacts. One of the features of such CM tools is that they allow developers[1] to *check-out* the artifacts they want to work on from the *public repository* that is accessible to all the participants and then the developers work on the checked-out artifacts in their own *private workspace* that is not accessible to others. After finishing their works, modified artifacts are then *checked-in* to the public repository again by the developers. This much coordination is not always enough to get the smooth updates and versioning of commonly used artifacts. Besides the tools at disposal, developers often follow a set of their own methods that include formal and informal techniques of coordination and communication to avoid conflicts and mismatches that might occur as they check-out, modify and check-in their artifacts.

The study done for the paper included a moderately sized software development team at NASA/Ames Research Center. The team already had setup a well defined software process for a group of developers and verification and validation team distributed through out a building. The team relied on different software tools in their process, two of them being a CM tool and bug tracking tool.

Conversations (interviews), documents and observations based on *shadowing techniques* were used to collect data from the team in form of notes and the data were analyzed using grounded theory techniques. The result of the survey led the authors come up with a set of practices (*articulation* work) that developers adopted in four situations: private work, in transition form private to public, public and in transition from public to private as tabulated below[2] –

Practices	Situations	Problem addressed
partial check-ins	Private work	to reduce 'back merges' resulting from parallel work on same documents
holding onto check-ins	Private to public transition	To minimize the disturbance to other's works
problems reports crossing team boundaries	Private to Public transition	To facilitate the management of interdependencies between the work of

[1] The terms Developers here might be equally pertaining to other stakeholders like members from the validation and verification team

[2] The table does not include practices during the situations *public work* and *transition from public to private* as those were more or less supported by the existing tools and process.

		different team members
code and design reviews	Private to Public transition	To prevent changes in the code *break the architecture*
“speeding up” the process	Private work	To avoid *merging*
the convention of adding the description of the impact of the changes in the e-mail sent to the group	Private to public transition (before check-ins)	For awareness of changes and informing about possible impacts the changes during parallel development will have. Also used as a learning mechanism for new developers as messages helps to identify experienced/expert persons in the team

The authors argue that these issues need analytical attention and computational support in order to enhance the cooperative work practices of developers in an organization.

Analysis of the paper

Most of the corrective measures project members (in the case studied in the paper) were taking seems to be occurring during the transition phase from private to public workspace. This is plausible as this phase accounts for all the modifications made in a users private workspace to be committed to the public workspace.

Using event notification servers to increase the awareness among developers about the ongoing concurrent changes in the program code seems to be a viable concept but as it is mentioned in the paper itself current CM tools tend to be independent of programming languages. This imposes a constraint on such systems, as fine grained monitoring of a localized context in a shared piece of code being concurrently modified won’t be possible unless the monitoring entity (or agent) is aware of the semantic and syntactic concerns of the code it monitors. The issue deteriorates when not only the source code but other artifacts like design documents, architectural artifacts and requirements specifications need to be monitored and kept synchronized. These documents even lack the formal and semantic representation that programming languages have. In such cases domain/language specific implementations for event monitoring and awareness might help but a general solution to such problems yet seems to be a great challenge to develop CM tools for collaborative software development in distributed environments. Furthermore, the monitoring needed to be done in background to constantly track changes in such tools still is computationally expensive.

Another issue worth noticing is that the team that was studied in this problem was located in the same building and developers and team members even had easy access to each other in terms of communication. Strategies like email notifications worked as people knew each other and were accessible. There was a centralized control on the project and there were managers aware of inspections and design violations in the changes being made. It might be an even greater challenge to consider collaborative issues in massively distributed projects with no central management as seen in many open source projects. Studying work practices in such projects might give additional hints in getting towards a better solution to the problem.

Unless it had been stated by one of the authors during his ICS 221 presentation that this paper was especially written with a CSCW perspective I would have been wandering why this paper is under the topic of *Field Study in Software Engineering* as the paper does not go into the details of how field studies are done and how theories are formulated from the empirical results. However, the paper and the presentation contained some hints that grounded theory was used for data analysis to come up with results. Still the paper addresses issues in collaborative software development and configuration management more than issues regarding field research.

ICS 221 Topic/Paper summary for the ninth week

Submitted by: **Sushil Bajracharya** (sbajrach@ics.uci.edu)

Topic: Harnessing Event Based Architectures

Paper selected for the summary: [HR98] Hilbert, D., Redmiles, D. An Approach to Large-Scale Collection of Application Usage Data Over the Internet, Proceedings of the Twentieth International Conference on Software Engineering (ICSE '98, Kyoto, Japan), IEEE Computer Society Press, April 19-25, 1998, pp. 136-145.

Summary of the Paper

This paper discusses a framework for a large scale collection of usability data using the internet in applications with event based user interface system. Three central themes of the paper are

- How the usability data collected from the users can be used to enhance the application development by considering those parts of applications users more concentrate on?
- How the events occurring in applications can be categorized and abstracted to define higher level of events by combining lower level events so that the amount to data to be sent through the network can be reduced, and
- How the monitoring code that deals with usability testing can be developed independent of the main application and yet can be made to work to instrument the application usage in tandem with the main application?

Usage expectations is the term given to the assumptions that software designers or developers make regarding the manner in which the software will be used by an end user. GUI software give wide range of possible usage opportunities to its users. The mismatch between the usage expectations by the designers and the actual usage by the users might make the performance inefficient and result in an overall bad user experience. To detect these possible anomalies in usage, usage expectations are modeled as *agents* that monitor the application being used. These agents are able to detect any usage mismatches or violations and take further actions like sending a report or informing the users of such mismatches. Expectation driven event monitoring system (EDEM) is a prototype tool that is mentioned in the paper that includes tools for defining such agents. EDEM also gives facility to create and edit agents to model a usage scenario. Results generated by the agents can be sent via email to a destination where they will be analyzed.

An event monitoring framework is described that defines three major *roles* - probes, distributors and consumers, and four *activities* - observation, processing, notification and actions[1]. A distinction between activity space (consisting objects and activities of interest in the system being monitored) and event space (made up of events and entities that correspond to objects and activities in the activity space) is made to represent different levels of abstractions among the events so that what is occurring in the system can be distinguished from what is made visible to the outer world. Agents can monitor the events as well as the other agents. A multi level event model makes it possible to define and generate higher level events by combining lower level events. This helps to cut down the amount of data to be sent through the network and makes possible large scale collection of usage data possible over a platform like internet.

Another section in the paper explains how the agent framework can be integrated to work with the GUI applications being monitored. A naming scheme for the components is used so that the components can be identified from their hierarchy. Using technology like JavaBeans, agent

[1] Description of these terms skipped for brevity

triggers are specified in terms of patterns of component events and agent *guards* are specified in terms of predicates involving component properties. Agent *actions* are even able to invoke the component methods if required. Integrating agents with the application is made seamless by a consistent naming scheme and defining agents in terms of user interface components and events. The agents live in a server that can be downloaded to the user's computer and then the agents can start monitoring the usage. Amount of data collected by the agent is controlled by the level of abstraction defined for the events to be watched. The data collected is sent back to the development server via email where it would be analyzed.

Analysis of the paper

I feel there are more issues remaining to be described if we are to take this paper as a description of a *framework* for event based usability monitoring. Only two of the important issues are discussed to a satisfactory depth in the paper, first it gives a layout for designing agents and second it gives an event monitoring framework.

It is not clear how the agent based software monitoring system is independently developed and seamlessly integrated with the application being monitored. The paper does mention few things like using component technology as JavaBeans, automatic download of agents from server to applications, requirement of inserting just few lines of code (two methods) in making the agents work and it further says other details are out of scope. But, I think if a framework is being discussed with monitoring as one of the important concerns, it would have been better to include some description of the technical solution that was envisioned to achieve a clean separation of monitoring concern from the main application unless such discussion would be totally redundant or have been entirely covered in some other documentation.

The process of analyzing the usage data is manual as all the reports are sent through email. It is not clear how such usage data are analyzed, for instance, are there any scientific techniques like statistical analysis or is it just the designer's intuition that leads to design changes. In case of massive data collected from thousand of users what techniques are there to deal with that mammoth data set?

There are two issues of concern in this paper. First is technological that defines how the entire framework for capturing events and generating meaningful information out of such events. Second issue is that related to human computer interface or of social-technical importance.

There might be several problems when implementing an application that constantly monitors the user's interaction with it. It might be difficult to convince the users that such background monitoring will not affect the performance of and hinder the normal work processes. Since agents independently collect information from the application and sends it over the network for analysis, users might not feel secure about information being sent about their using the application. In this regard there must be techniques that assure the users that no security and privacy breaches are being made while doing so.

ICS 221 Topic/Paper summary for the eight week

Submitted by: **Sushil Bajracharya** (sbajrach@ics.uci.edu)

Topic: Product Line Architectures

Paper selected for the summary:

[HMR+01] André van der Hoek, Marija Mikic-Rakic, Roshanak Roshandel, and Neno Medvidovic, *Taming Architectural Evolution*. In Proceedings of the Sixth European Software Engineering Conference (ESEC) and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), Vienna, Austria, September 2001.

Summary of the Paper

This paper presents a system model and description of a software architecture design tool called Mae that supports architectural evolution. The underlying model for the tool that manages the evolution of software architecture come from two areas of software engineering, configuration management (CM) and software architecture.

The paper starts by giving an overview of software architecture, discussing basic elements of software architecture like *components*, *connectors* and their interconnections, architectural *styles* and architecture description languages (*ADLs*). Then, three generation of CM systems are presented mentioning important concepts such as *version trees, configuration and configuration specifications*, *repository* and CM *policies.* Rest of the paper describes the system model for Mae, its implementation and capabilities.

The architectural system model for Mae distinguishes between *types* and *instances* of the elements[1] that make up the model viz. *Components*, *Connectors* and *Interfaces*. *Interface* types are the basic units that define the system model for the Mae tool. All other components or connectors are defined on the basis of a base interface type or some other element types that are eventually based on interface types. Interface is at the top level of the definition hierarchy and versioning of architectural elements is done at the *type* level. A V*ariant component* type is defined when in a design one component can be chosen from a set of many components. A version given to a component or a connector *instance* is the version that it inherits from its *type*, this setup makes the versioning process tractable and simple. An element type in the model can have one *Ascendant* from which it derives and many *Descendants* that further list its child elements. All the types are uniquely identified with a *Name* and *Revision* attributes. A *Representation* field in a type allows architecture style specific details to be included in a type definition. A component type further contains attributes like *Constraints*, *Behavior*, *Style* and *Sub Type* for architecture specific adornments. What distinguishes a *Variant* type from a component type is the *Component* (list of component instances) and *Variant Property Name* field (that takes in a variant property value designated as *Method,* responsible to instantiate one of the component instances from the C*omponent* list).

Instances are simply named instances of an element *Type.* In this regard what makes up a software configuration is a collection of different element *instances* each belonging to a particular *type*. Interface instances have *name*, *direction* and *interface type* descriptions. Connector instances are different from component instances (that have only the *name* and *component type* or *variant component type* listed) as connector instances have the associated set of links that connect sets of component instances.

[1] Element refers to architectural elements throughout this document

Mae has enough flexibility with its attributes so that a generic architecture can be defined that can be later specialized to a particular style description (for example, the system model supports connector instances that link component interfaces, both directly and via the connector's interfaces). The connectors and component instances of the elements put together define an architecture instance. Properties like ascendants, descendents and revision helps to make up the version trees for the elements (giving *linear* and *diverging paths of evolution*). Attributes like style and sub type control architectural constraints like rules of composition for elements and interface/behavior conformance. Furthermore, behavior and constraints properties are used to check the *compatibility rules and expectations of component and connector types*.

Based upon this system model, Mae is a software architecture design tool that allows architects to visually develop architectures with versioned elements (components and connectors). It maintains a change history for each element being modified and has a support for a kind of parameterized configuration of architectures as it lets the architects select different property values for variants and other undefined options to come up with different configurations of elements and eventually different architectures (simply by setting those values). While doing so the tool also is capable of determining the consistency and validity of the architecture being developed by querying the different values set to the attributes that element instances have. Once an architecture version has been developed Mae can generate a style specific architecture description (C2SADEL) and perform some style specific analyses.

Analysis of the paper

Elements of software architecture make the foundation upon which Mae is based. Though there is an introductory section on Software Architecture in the paper, there is no description or example that describes how the attributes pertaining to architectural constraints like *Sub Type*, *Behavior* and *Constraint* are used in the system. More elaborate description of these terms or some examples would have explained how the system model is able to take control of core architectural aspects of the evolution using these attributes.

Looking on the CM aspect of the system I do not see what kind of policy support is there in the tool that manages advance CM issues like concurrent collaborative development of architecture in the Mae environment.

While enhanced design experience, multi-versioning connectors and automated change script generation are exciting features of Mae, I find some other issues relevant to further enhance such a tool.

- There in not any insinuation about how well the system as whole is able to get a hold of the architecture and implementation together. Automatic code generation for template classes is mentioned as one of the features but it is unclear how exactly the system will (or will not) support the synchronization of the architecture or design with the implementation as changes are made in the implementation.

- One of critical aspects that make controlling the evolution of software artifacts difficult is the subtle issues that cross-cuts different components and cannot be encapsulated in a single module. Such issues have been recognized as *Aspects* at the implementation (programming) level and techniques like Aspect Oriented Programming have been developed to tackle such problems. System model like that of Mae which has a solid architectural foundation should be augmented to address such cross-cutting issues (*Aspects*) at the architectural level to manage the interdependencies among architectural elements as these *Aspects* certainly are one of the factors that hurdles a smooth evolution of software artifacts including the architecture.

ICS 221 Topic/Paper summary for the seventh week

Submitted by: **Sushil Bajracharya** (sbajrach@ics.uci.edu)

Topic: Open Source Software Development

Paper selected for the summary:

[Sca02] W. Scacchi, Understanding the Requirements for Developing Open Source Software Systems, IEEE Proceedings--Software, 149(1), 24-39, February 2002

Summary of the Paper

[Sca02] presents a comparative and descriptive view of the requirements engineering processes in open source software communities as a result of empirical studies in different open source communities. Four different categories of open source software development communities were studied viz, i) networked computer games, ii) internet/web infrastructure, iii) X-ray astronomy and deep space imaging and iv) academic software design research. The motivation for taking these variants into consideration was to investigate these communities from an *ethnographic perspective* with an assumption that *no prior established accepted framework* defining how software should be developed exists in such communities.

The author uses seven different empirical field study principles for his research, they are - i) The *hermeneutic circle* (analysis of the evolution of the whole process from its parts and interaction between the whole and the parts), ii) *contextualization* (need to identify the context that characterizes social and historical background of the process/research topic) , iii) *revealing the interaction of the researcher and the subjects/artifacts (researcher as a participant observer),* iv) *abstraction and generalization* (generalization and summary of research findings across many similar communities) , v) *dialogical reasoning* (comparing existing methodology with that found empirically through *participant observation*), vi) *multiple interpretations* (need of the realization that different participants *see and experience things differently*) and vii) *suspicion to possible biases or systematic distortions* (necessity to look for alternative explanations of the data from the empirical findings and to welcome unbiased and different views on the topic).

The research based on these principles summarizes the differences between the requirements engineering practices in the open source software community and the standard/classic software engineering process as follows:

Stages in Standard SRE[1]	**Open source software communities' practice**[2]	**Remarks**
Elicitation	Assertion of requirements	Requirements are asserted as system capabilities
Analysis	Requirements reading, sense-making, and accountability	Descriptions are not self-complete, readers make their own effort to understand and gather incomplete information
Specification and modeling	Continually emerging webs of software discourse	Functional and non-functional requirements continuously are refined via narrative descriptions supported with domain specific artifacts
Validation	Condensing discourse that hardens and concentrates system functionality and	Requirements co-evolve with design and implementation. No formal validations but validations through consensus among

[1] SRE = Software Requirements Engineering
[2] Alternatives to the standard SRE practices

	community development	participants
Communication	Global access to open software webs	Hypertext style organization of requirements that is accessible globally via platforms like the World Wide Web

Based on these differences the author suggests the need of characterizing a common foundation for the development of open source software requirements. He presents a notion of *software informalisms* to collectively describe the Web-based descriptions of all the functional and non-functional requirements for open source software systems as discovered in the stages listed above in the table. Eight such software informalisms are listed such as - usage of computer based communication tools for community communications, using linked web pages as scenarios of usage, How-To guides, external publications in the domain, various interlinked web sources etc.

Development of open source software system requirements is a community building process, a *complex web of socio-technical processes*. It is a collaborative effort based on computer mediated communication that forms a complex social structure in the cyberspace. The author states that since in open source software systems, the developers are generally the end-users of the systems (unlike in commercial/traditional systems), open source software can suffice with reliance on these informal and light weight or non rigorous software informalisms.

The author concludes by raising few other questions regarding this informal nature of requirements engineering process in open source software systems.

Analysis

I think the benefit of studying this paper is twofold. It gives a good overview of comparative study of requirements engineering practices in the open source software community and also demonstrates how an empirical study can be made in such a virtual community in cyberspace following the seven different empirical studies laid out in the beginning of the paper. This can be a valuable guideline for somebody trying to conduct similar studies especially in virtual communities, glued together by modern communication mechanisms like the World Wide Web, e-mail, bulletin boards and threaded discussion boards, where participation and field work have different meanings than in the real physical workspaces.

Mostly the software produced in the open source community is generic and targeted for a community of users. Unlike a customized software product for a particular customer, software products coming from the open source software communities are envisioned by domain experts or lead researchers in the field (as in the X-ray astronomy and the academic research). So the problem of impedence mismatch between a potential not-so-technically-sound customer and developers, as frequently occurring in commercial software requirements engineering stage, never occurs in the open source software requirements engineering stage. Rather, as the author mentions, the problem can be of opposite nature. The developers know exactly what they want to build or they might come to a consensus about what to build, but in highly specialized domains as in the X-ray astronomy, the developers (like astrophysicists) might need to adequately train themselves in the software development task to tackle the problem of how to build something they want.

Observing current categories of software being developed in the open source software community the author's argument about software informalisms sufficing the requirements seems plausible. But is this an implication that mission critical software (like those in health care, nuclear reactors) that must undergo rigorous treatment during specification cannot be developed in a community like open source software community?

Again, how feasible is that a non technical customer can approach the open source software community to build him/her a customized software product?

ICS 221 Topic/Paper summary for the sixth week

Submitted by: **Sushil Bajracharya** (sbajrach@ics.uci.edu)

Topic: Software Design Environments

Paper selected for the summary:

[RR98] J. Robbins, and D. Redmiles. "Software Architecture Critics in the Argo Design Environment". Knowledge Based Systems, 1998, pp. 47-60.

Summary of the Paper

Software design has been established as one of the most important and prioritized task in the software engineering life cycle. There are various commercial design tools available that are developed to automate and ease the design process. Most of such current tools focus on contemporary tasks like design diagramming, modeling, refactoring, reverse engineering and automatic code generation. To make them more appealing many tools support more than one implementation platform. But, these tools fail to address more human aspects of design such as decision making and other cognitive challenges designers or architects face during the design.

In [RR98] the authors present Argo, a software architecture design environment that goes beyond supporting conventional design tasks with its *critiquing infrastructure* that *supports decision making by automatically supplying knowledge that is timely and relevant to decisions at hand*. One of the main features of Argo is its ability to assist the designers in making decisions even with an incomplete set of information or knowledge available. This is different from other tools that suggests design changes or make optimizations only after at least one complete cycle of the design process. Handling such partial information directly supports the cognitive theory of *reflection-in-action* that states *designers can best evaluate their designs while they are engaged in making design decisions, not after.* Argo implements software agents acting as *critics* that inform the architect about possible issues like changes to be made or probable violation of design constraints that are pertinent to current design modifications being made. The critics do not make the changes or implement the advices they have regarding the current state of design but rather they make the architect aware of such issues by listing and organizing them in a multi-view model of a *to-do* list. Since critics are constantly watching the modification going on in the design/architecture, voluminous feedback data might be accumulated in the to-do list. Also, the items that appear on the to-do list should be customizable by the architects according to the context of design phase. To manage these issues Argo has a *Criticism Control Mechanism.* Critics are pessimistic by nature in suggesting design changes, so to make the architect comfortable with this criticizing behavior of various critics built inside Argo, the to-do list is implemented as a dynamic *Feedback Management* tool. Argo also supports a *Corrective Automation* that provides an automatic fix to some design problems that the critics can identify as having specific automatable solutions. Architects interact with the items in the to-do list instead of the critics themselves. Architects may take their decision as per the critics' suggestion or they may apply their own heuristics and knowledge and deviate from such suggestions. In any case all the interactions architects make are stored in a *Design History*. This history can serve as a rationale for future inferences.

Users of the Argo follow the ADAIR (*Activate->Detect->Advise->Improve->Record*) critiquing process to work through the whole design process. *Activation* enables a subset of appropriate critics that *detect* opportunities for possible suggestions according to the modification being done on the design/architecture. Critics *advise* architects about possible deign problems and improvements. The architect may or may not follow the advice given on the basis of the improvement to be done in the architecture any decision made by the architect is *recorded* for future reference.

Rest of the paper gives implementation details of the various components of Argo and also provides a comparison study of Argo with similar tools.

Analysis

Reading the paper I did not exactly get which feature of Argo exactly addresses the cognitive theory of *comprehension and problem solving*. As far as I could understand the to-do list can be viewed from multiple perspectives and I think this might be one way of representing multiple mental models of the design. It would have been a lot easier to comprehend if there had been a short summary in the paper that listed all the features in Argo that mapped to the respective cognitive issues.

As far as the paper describes Argo seems be a tool targeted towards single users. Software development is a task that involves contribution from various people and stakeholders in all steps and these people need to be openly communicating with each other. Argo does not seem to address the collaborative support that a tool needs to provide to its users especially in a domain like software engineering. It is very possible that more than one designers are working on a software project in separate parts, even though Argo has a feedback recording mechanism and it maintain a design history, it is not very clear what degree of collaborative support can it provide. There is a support for *email expert* dialog box that helps architects to query the maintainers of the critic but as the paper says making tool like Argo a platform for capturing and reusing organizational expertise will add more value to it.

I do not see how the knowledge built within a critic can be assessed to be actually valuable or not. There does not seem to be any evaluation technique that actually measures the effectiveness of the knowledge that is being accumulated inside the tool.

Conclusions

Tools like Argo support its users by considering several cognitive issues that may arise while they are being used. Building knowledge support inside design tools will not only make design task more effective and cut down the software cost by being able to detect design flaws early in the process, it also indicates a beginning of a family of tools that can serve as expert systems that constantly guide the users and make them better learn the tool as they use them.

ICS 221 Topic/Paper summary for the fifth week

Submitted by: **Sushil Bajracharya** (sbajrach@ics.uci.edu)

Topic: CSCW (Computer Supported Cooperative Work)

Paper selected for the summary:

[A000] Mark S. Ackerman, *The Intellectual Challenge of CSCW: The Gap Between Social Requirements and Technical Feasibility*, Preprint to be published in Human-Computer Interaction.

Summary of the Paper

This paper [A000] is a dense survey of research findings in CSCW. The author brings in the term *social-technological gap* that remains as the core issue being discussed in the paper. Social-technological gap is described as the inability of current technology to provide all the computational support needed for social interactions, for example, *the lack of computational support for sharing information, roles and other social policies*. First, the paper presents several important findings of technical research in CSCW and put forwards the issues to be addressed that give rise to this social-technological gap. Secondly, an example is presented to demonstrate to show why such a gap[1] is inherent and finally solutions to the social-technical gap are discussed and the author puts forward the *science of the artificial* as an important viewpoint for resolving the gap.

Some of the points discussed as a summary of findings in CSCW research are as follows - i) Systems[2] often fail to handle the details of interactions as human do and systems cannot be context sensitive as humans regarding shared understanding of information, ii) CSCW attempts to resolve issues related to the conflict and coordination, exception in work processes, awareness among users and trade-offs users make in sharing information, iii) People adapt to the systems and sometimes they want or make the systems adapt to their preferences and iv) The incentives for the users as a reward for their collaboration and coordination is a critical issue for maximum usage of systems. In working/commercial systems some of these issues are sacrificed and *such trade-off introduces the tension in those implementations between "technically working" and "organizationally workable" systems*.

An illustration of the social-technical gap is the P3P[3] project. Reasons that make the social-technical gap inherent in such an online privacy initiative are due to the fact that systems cannot be designed according to the way humans work. Three such limitations systems have in such context are - i) Systems are not able to make fine-grained distinctions based on context and previous knowledge, ii) Systems are not socially flexible, the ways humans work with multiple roles and switch their states based on conditions is difficult to be exactly reflected as a system behavior and iii) Humans are able to deal with indeterminate situations and they can perform their role with a certain level of ambiguity. Systems cannot work with such level of ambiguity.

The remaining section is a not so brief analysis of the social-technical gap and solutions to it but the overall concept can be summarized as follows - i) Technical researchers in CSCW are aware of the gap and understand its nature, ii) Solutions to resolve the social-technical gap by technical advancements alone like advancements in the area of machine learning and neural networks is unlikely, iii) Users should not be forced to adapt to a system, iv) The solution to ameliorate this gap lies in considering CSCW as *a science of the artificial*. CSCW should be treated both as an

---

[1] The terms 'social-technological gap' and 'gap' are interchangeable in this summary
[2] System(s) in this document refer(s) to those systems used as CSCW tools (as discussed in section 2 of the paper)
[3]P3P is a standard from W3C that aims to provide *a simple, automated way for users to gain more control over the use of personal information on Web sites they visit. At its most basic level, P3P is a standardized set of multiple-choice questions, covering all the major aspects of a Web site's privacy policies.* http://www.w3.org/P3P/

engineering discipline and as a social science. This can give rise to new educational techniques that teach computer scientists *organizational and social impacts that result from their designs* and teach them to build systems to support the social worlds if not then teach them what is possible and what is not. Two important steps for working towards the solution are

- *First order approximations* – having solutions that partially solve problems with known trade offs. Like having a software critic that would guide the user taking decision rather than the software taking decision by itself.
- *Determining guiding research principles* to designing around the gap which is more difficult when a discipline of science is still nascent.

Analysis

No doubt this paper is a good survey of CSCW literature. By and large the gist of the paper can be well construed. But numerous references to publications make it difficult to grasp all the facts being discussed at first. One cannot however complain about such succinctness and implicit remarks authors make in research papers but especially the sections that discuss the findings of the survey mentioning terms like 'Boundary objects', 'activity breakdowns' and discussions that refer to topics like 'fluid dynamics', 'neo-Taylorism' and 'Simonian philosophy' definitely requires the reader to be aware of such theories beforehand in order to get the exact point the author is trying to make in those sections of the paper.

I agree with the author when he explicitly states that the survey is biased, as it discusses CSCW research from a technical perspective. I think it is more focused on issues pertaining to HCI (Human Computer Interface). The findings that are listed are mainly of those from technical research. The author certainly highlights the necessity of interdisciplinary focus needed for CSCW research and argues around theories like *science of the artificial* but issues related to studies of social science, economics and organizational science are not explicitly visible in the paper. Also, the view of CSCW from the user's eye seems to be worthwhile being included in such discussions that involve the study of human and social behaviors in conjunction with technical advancements.

Conclusions/Comments

Human thinking and behavior is a complex process. There still are unknown facts about the way we perceive things and make our decisions. The way we interact with each other and the way we form and participate in a society and take initiatives to solve problems in hand is driven by different factors including but not limited to knowledge, experience, environment, temperament, emotion, cultural beliefs, tradition etc. Furthermore every person being different she/he interacts in a unique manner that reflects her/his own idiosyncrasies even in similar conditions and given similar tasks. Surely actions are governed by rules and these rules can be imposed in a prescriptive manner by software or similar tools but given a multitude of options and scenarios, there can always be a case when all such scenarios cannot be dictated by a defined set of rules. And yet again we cannot represent all the humane factors in a system.

Computers and other tools of trade definitely help to ease our task in terms of data processing and storage but such a tool would be absolutely impeccable only if it would work the way we want or more precisely the way it *must* work in a given situation. With new technology and inventions these tools have been augmented to the next level, as we can see a simple PC has grown as a powerful machine able to work in interconnected and distributed environments but these advances simply are manifestations of technological advents. In other words we have been making our tools powerful in terms of their own machinery capabilities (of crunching large numbers or handling high volume of data) only. Today we have at our disposal a machine that can execute millions of instructions per second and have gigabytes of memory, in past such machines were a factor of times less powerful in terms of these metrics. With advent of internetworking platform like the Internet and open platform for collaboration like the Web we have this tremendous opportunity to develop tightly integrated tools and setup virtual societies and organizations in the cyberspace. But this advancement does not touch upon the human aspect of solving problems. In this regard the machines we have today work the very same way they used to do in the past. Our impulsive nature is way too complex to be exactly mapped in these deterministic problem solving machines and tools. The interdisciplinary nature of CSCW research is a consequence of the fact that now we are seeking novel solutions to the problem which in the author's words is the problem of *socio-technological gap*.

Studying and inventing the techniques that will just make our computing machines work faster or handle larger amount of data is not enough to make them ideal tools for CSCW. Adding interconnectivity among these machines facilitates tasks but they just perplex us with high volume of instantaneous data or problem sets. I hope interdisciplinary research that brings together various realms of human thoughts, behavior and workings in standalone and collaborative social setting will be an attempt to solve problems and to make our life easier dealing with the *social-technological gap*.

ICS 221 Topic/Paper summary for the fourth week

Submitted by: **Sushil Bajracharya** (sbajrach@ics.uci.edu)

Topic: Aspect Oriented Programming

Paper selected for the summary:

G. Kiczales, J. Lampoing, A. Mendhekar, C. Maeda, C. Lopez, J. Loingtier, and J. Irwin. *Aspect-oriented programming*. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1261, 1997

Summary of the Paper

This is ***the*** seminal paper that puts together the fundamental concepts of Aspect Oriented Programming (AOP). A basic framework for AOP is presented that is illustrated with few examples highlighting two motivating factors for AOP – i) *ease of program understanding* and ii) *efficiency of the program/software*.

Almost always software of considerable size and business value is an evolving product. The size and the complexity of the software and the interdependency of modular components that make up the software as a whole increases as the software evolve. Software engineers at their disposal have design guidelines like abstraction, information hiding and modularity that attributes to high cohesion and low coupling of components eventually aiming to produce a tractable and flexible software product. Even with these design tools at disposal there remain subtle implementation issues that bind these components together in a cohesive whole. These issues (such as *concurrency* and *memory access patterns,* especially perceptible during runtime) are not prone to the conventional functional decomposition techniques. These issues *cross-cut* software components and stand out as the *properties that affect the performance or semantics of the components in systematic ways.* The authors define such issues as ***aspects***.

The principal goal of AOP is to provide mechanisms to allow separation of these *aspects* from ***components*** (*a cleanly encapsulated unit*) in a system. This separation results in an easy to understandable and an efficient program. To achieve this, the framework for AOP comprises of i) *Component language – to program the components*. This is like other high level languages that allow us to code programming constructs like procedures, classes etc. ii) *Aspect language – to program the aspects.* Component language and aspect language give component program and aspect program respectively when implemented. iii) *Aspect weaver* – this is responsible to process the component and aspect language together to make then work.

Component programs tend to be simple as they *cannot preempt anything aspect programs need to control*. For example if Java is being used as a component program and if concurrency is one of the aspects then using keyword like 'synchronized' and methods like 'wait()' is not allowed while writing the component code. This should rather be addressed by the aspect program. Another important thing to notice here is that the aspect weaver does not work as an optimizing compiler in producing efficient code by combining the component and aspect language. All the possibilities for optimizations should be already represented by the aspect language in the aspect program. The weaver simply integrates programs written in component and aspect languages. One subtle thing about weavers is that they have to take into account the presence of *join points*. Join points are *elements of the component language semantics that the aspect programs coordinate with*.

In this way separation of concerns and introduction of aspects is a promising way to achieve understandability and efficiency provided that enough care is taken to *explicitly* program all the required details as aspects.

Analysis

The paper does a good job in laying down the foundation for AOP but leaves behind many open questions regarding implementation. The paper does not establish a standard implementation technique for AOP and its three constituents. Instead it discusses few problem specific implementations in the examples and hints that the implementation can be according to ease and the nature of the problem being solved. For example one of the examples in paper says that the weaver gave a C program as an output. That was a case of compile time weaving. We can even have a running virtual machine or an interpreter that works as a weaver, weaving aspects and components during runtime.

Further issues

Aspect Oriented Programming has come a long way in a short period of time. As Prof. Lopes appropriately put in her words during ICS 221 lecture, the project 'took off'. There has been a widespread adoption of AOP as a programming paradigm that has contributed to its evolution.

The original discussion of AOP in the paper considers both domain specific and domain independent aspects. Most of the aspects that result in enhancing the efficiency of the program seem to be domain specific. Coming up with such aspects might entail a significant level of expertise in a particular domain. This opens up a possibility of coming up with expert collections of domain specific code enhancements as aspects. This seems analogous to the concept of design patterns that enable software designers to encapsulate efficient design solutions in well defined patterns.

The current frameworks for AOP seem to be more generic in nature and seem not to explicitly take efficiency of the program into consideration. Currently AOP is primarily being used as a mechanism to add additional features to software components in-the-fly during runtime. As an example the JBoss[1] AOP framework enables programmers to write simple Java classes (components) and then add features like transactional caching in those simple components by 'inserting' them into the JBoss engine during runtime. This dynamic runtime behavior is a highly desirable attribute in today's software environment as it enables lot of flexibility regarding what we want to do with our component and where we want to execute them. This certainly makes the life of the programmer writing the simple component easier. It also succeeds in separating out a major chunk of concern from the code but, the efficiency concerns are not quite explicit in such AOP frameworks. And, most of the performance issues will be driven by the aspects that enable such dynamic behaviors.

Moreover, most of the implementations for AOP today take a high level language like Java as a component language and then use extensions to that language to define aspects. They utilize mechanisms like reflection in the (runtime) weaving process. For example again in case of the JBoss AOP framework, they use Javassist, a Java bytecode manipulation framework to intercept method calls and to achieve all the on-the-fly transformations they want. But surely there are overheads going on in terms of additional objects being created and additional tasks being performed during runtime. Tradeoffs like these should be well considered while using such an implementation.

Much of the implementation issues regarding aspects (like loop manipulation for efficiency) that make AOP work is discussed at a programming language level in the paper. This gives a feel that AOP seem to best suit 'programmers' than 'designers' or 'architects'. It will need very good programmers to write appropriate aspects in the aspect language if one needs to consider the efficiency to be brought into the final code. Eventually how AOP relates to and effects other

[1] www.jboss.org. JBoss is an open source J2EE server.

software engineering tasks like design, testing, specification and even collaboration might be an area we need to look into to see how AOP fits inside the big picture of software engineering.

Among one of the noticeable challenges to developers adopting AOP currently might be the reluctance they face fom the management for embracing a new technology. There still are not appropriate metrics that enable one to prove that what efficiency or performance benefit was achieved with AOP.

Conclusion

Concepts like *separation of concerns* and *functional decomposition* are not new. They were pioneered by the grandmasters in the field of software like Dijkstra and Parnas. These concepts have heavily dominated the programming methodology we have adopted and directed the way we produce software today. Even though we have been thinking the very same way for long, that does not mean there cannot be alternate or even better ways to rethink. AOP brings into light a new perspective of problem decomposition and software organization. I believe AOP is not an attempt to replace our current practices in software development rather it adds to and complements our current practices in the field. I think it will make the same impact 10 years from now that Object Oriented Programming has made since its widespread adoption in the area of software engineering.

ICS 221 Topic/Paper summary for the third week

Submitted by: **Sushil Bajracharya** (sbajrach@ics.uci.edu)

Topic: Software Requirements

Paper selected for the summary:

[Hen80] Kathryn L. Heninger. "Specifying Software Requirements for Complex Systems: New Techniques and Their Application". IEEE Transactions on Software Engineering, SE-6(1):2-13, January 1980.

Summary of the Paper [Hen80]

This paper gives an outline for *a disciplined approach to requirements specification*. Three major issues that are discussed under this are the *objectives,* the *design principles* and the specific *techniques* that constitute the methodology for developing requirements document. The techniques are illustrated with an example to develop a requirements document for the embedded software of the A-7 aircraft.

The *objective* follows the universal principle in requirements gathering i.e. capturing questions that answer WHAT but not HOW. It emphasizes on capturing constraints specially those pertaining to *hardware interfaces.* The techniques described are divided into four sections, techniques for specifying *Hardware Interfaces,* for specifying *Software Interfaces*, for specifying *undesired events* and for *characterizing types of changes*[1]. These techniques are guided by the *design principles* like *separation of concern* and *stepwise refinement* that are seminal concepts in program design.

Hardware interface description is organized by input and output *data items*. Special *forms* are devised to capture all possible information about such data items. These forms are able to capture enough information that can describe input data items as the *resources available to solve the problem* and output data items in terms of the *effect they will have* on associated hardware components. There is a provision to use mnemonics to refer to all the data items in the system which follows a special bracketing pattern that distinguishes their type. To describe values of similar types a parameterized definition is used. All this setup makes the interface description for hardware consistent and complete.

Software is described as a set of *functions* that determines values for one or more data items. Instead of having a direct mapping between input and output values, output values are expressed to be determined by a function of *conditions* (set of predicates) and *events* (that occur when conditions change). A notion of *modes* is introduced that basically are collections of conditions that describe a particular state a system is in. Special notations are followed to represent events, conditions and modes. There are *condition tables* and *event tables* that can group together different modes and show the transition between modes. These tables also show the relationships between modes, conditions and events. This setup is formal enough to capture the precise behavior of the system and check if all the required conditions are met or satisfied. *Function forms* are used to specify system behavior by putting together the modes, data items and tables together. This constitutes a logical unit that can be traced to check the desired system behavior/requirements concisely and correctly. Furthermore, *text macros* are used to abbreviate descriptions and make things concise.

Overall, the paper addresses the major problems in working with requirements documents such as – ambiguity, redundancy, incompleteness, inconsistency and the inability to be verified. The paper contributes in giving directions toward alleviating these problems by using good design principles and concise formal techniques. It also outlines a table of content for a requirements document that can be used as a template to write requirements document for similar projects.

[1] last two techniques are dealt lightly and informally

Analysis and further issues

The approach towards requirements document presented in the paper applies well when there are many technical details to be taken care of such as hardware constraints and sensor inputs that deal with critical issues like weapons and human life. A requirements document however, can also be considered as a bridge of contract between customers and developers. When customers are not comfortable with notational brevity and tabular representation of the system behavior as presented in the paper, such requirements document will fail to serve as the contract document. In the paper there is no explicit method described to render the dynamics of the system such as information flow (what kind of information flows from where to where), especially to a customer who wants to perceive such behavior effortlessly. Techniques of analysis using Data Flow Diagrams or other semi formal notations like activity diagrams in UML seem to address these issues in a simpler manner.

This gives a feel that the techniques in the paper are domain specific, well suited for describing software systems that interact with hardware and sensors. For documenting requirements for information systems with complex information flow such as a hospital management system or an Enterprise Resource Planning application, techniques described in the paper will not entirely cover all the issues in requirements engineering. They can however be applied to some specific problem subsets in such software like specifying concurrency and performance requirements.

The entire process of capturing and maintaining the requirements document discussed in the paper is manual. No automation or special tool was used to implement the *techniques* presented. The necessity of tool support for such techniques is even greater when software projects evolve in their complexity. If there are more inconsistencies and mismatches between the requirements document and the implementation later in the project, requirements document is treated as a throwaway artifact as the implementation is the product to be delivered. The paper too describes maintaining consistency as an important aspect for the usability of the requirements document. It claims that the *function tables* are capable of detecting *inconsistencies that show up conspicuously*. But again the tables need to be manually looked up and maintained by hand for any modifications.

I would say the techniques described in the paper are formal enough for manual maneuvers but not formal enough to implement automated tools like automatic consistency checking and test case generation. There are not enough details that give a precise formal representation of items like *function forms*, *macros*, *data item forms*, *condition tables* and *event tables* introduced in the paper to develop such tools. It is quite inevitable that errors will be introduced and some of them will be overlooked when the documents grow bigger with hundreds (and may be thousands) of *macros*, *data items* and multiple *modes*. Techniques discussed in the paper can be made more efficient and error proof if the supporting tools for constructing different artifacts like forms, tables and macros are available and if such tools can maintain consistency and automate tasks.

Since the techniques described were applied to software that had already been built it was easy to ask the right questions and making right predictions about the possible change that will come in the future. For software that doesn't exist yet coming up with the correct questions (or even answers) all the time and making right predictions regarding possible changes will always be difficult.

Conclusion

Despite being manual, the process presented in the paper is *disciplined* as it follows a set of guidelines, concise notations and formal ways to document functional requirements. The methodology is convincing as the concepts and example goes hand on hand. Because of the fact that it so closely follows and is based on the requirements documentation for the embedded system software of the A-7 aircraft, it leaves an impression that such a methodology is restricted to or may be worth being practiced in such (mission critical) domains only.

ICS 221 Topic/Paper summary for the second week

Submitted by: **Sushil Bajracharya** (sbajrach@ics.uci.edu)

Topic: Interoperability and Middleware

Paper selected for the summary:

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, Volume 12, Issue 6, Nov. 1995 (pp. 17-26) November 1995.

Summary of the Paper [GAO95]

In this paper the authors introduce the term *Architectural Mismatch* and explain it to be the root cause of the problems they faced in building a tool by reusing already available software components that they believed would best fit their requirements. Even though they were trying to reuse *standard pieces of software*, considerable amount of work had to be done to integrate all those pieces. *Excessive Code*, *poor performance*, necessity of *reverse engineering* and modifying/fine-tuning the components, *unnecessary complications* and the interdependency and the conflicts in the build process due to the changes made were the major problems that had to be worked out while coming up with *AESOP*, the product described in the paper.

Viewing from an a*rchitectural* level a software product can be seen as a combination of *components* and *connectors*. The implicit assumptions designers/developers make while developing these *components and connectors* conflict when they are assembled and contribute to the A*rchitectural Mismatch.* This eventually causes all the problems mentioned earlier. The authors identify that possible *assumptions* are made regarding –

a. *The nature of the components* which again can be regarding – *the infrastructure* that bring in the unwanted code, the *control model* that often necessitates *reverse engineering* and code modification and the *data model* that bring about inconsistency in data format and exchange
b. *Connectors* with requirements that constrain reuse due to assumptions they make regarding the *protocol* and/or *data model* (format) they follow for communication between components
c. *Structure* these elements make when they will eventually be composed into an application, especially regarding the *presence/absence of a particular* element they might have and the way they will be communicate
d. *Construction process* that defines the dependency among different elements in the order they are instantiated and eventually the manner they are built (linked/compiled)

As a solution to avoid and plan ahead for this *Architectural Mismatch* following techniques are suggested –

a. stating and *making the assumptions explicit*
b. *using orthogonal subcomponents* - loosely coupled sub-components to enable later substitution
c. using *mediators* and *wrappers* as bridging techniques in cases of inevitable mismatch
d. using expertise to develop an efficient design methodology that spans multiple application domains. This will help maintain expert idea about the right composition of *architecture components*

It was possible for the authors to perceive these *assumptions* through an *Architectural lens.* So they suggest solution to these problems will be hard enough till there is enough formalism and methodology to express and design a software system with a standard *Software Architecture.*

Improvements in compiling and linking software modules will not be enough to alleviate this problem.

Applicability of the paper's ideas to the topic

The problem discussed in the paper is surely an interoperability problem. The solutions prescribed in the paper that advocates an architectural based design of software components is already seen in practice.

Middleware technologies are using most of the concepts as described in the paper to achieve interoperability among components. As middleware software work on a defined protocol and a standard it surely is evident that they make lot of the Architectural assumptions explicit so that all the elements in the system work without conflicts. Wrapping a component to export a negotiating interface to comply with other components are techniques Middleware use, this also has been discussed as one of the *bridging techniques* in the paper. Most of the popular implementations of Middleware are done using object oriented methods following well defined patterns established as 'design patterns' in specific domains. This also seems to be congruent with the authors' idea of *developing and using sources of design guidance*.

Remaining issues

Excessive code is mentioned as one of the problems the authors mention that bring down the performance of software composition but Middleware today still are add-ons to the system as either they provide extra wrapping or they act as *mediators* sitting in between and making two components communicate. There must be ways to keep performance at peak despite of this extra addition of *wrappers* or *mediators* and the extra processing being done.

There exist a collection of Architectural Styles and Architecture Description Languages (ADLs) that can well define or can be extended to define a Software Architecture at a desired formal level. But the abundance of these again might demand an agreed standard so that components following different ADLs/Styles can understand each others declarations about any assumptions or constraints they have.

It seems we do not yet have a cataloging system, taxonomy and metrics for Software Architectures. This might be an area to work on for developing proper *design guidance* techniques.

Metrics that measure the goodness of badness of architectural components as possible candidates for composition and, metrics that measure the efficiency of a particular style in composing elements together need to be developed.

Another interesting area to be explored might be coming up with techniques to record and use expertise that helps in coming up with an efficient composition of *Architectural Components*.

Conclusion

The paper gives an introduction to software architecture and discusses how an architectural view of reusable software components can identify the problems that might arise while building a system composed of such components. Solutions are listed to handle these problems due to this *Architectural Mismatch* but there still remain a lot of work to be done as a single agreeable standard for software architecture does not exist yet.

Summary of the paper **“Web-based Development of complex information products”** by Roy T. Fielding, E. James Whitehead, Jr., Kenneth M. Anderson, Gregory A. Bolcer, Peyman Oreizy and Richard N. Taylor

Submitted by Sushil K Bajracharya as ICS 221 assignment for Week 1.
(sbajrach@ics.uci.edu)

Summary

The paper discusses problems in leveraging the World-Wide Web (Web) as a true collaborative project work space and proposes enhancements that can uplift the Web’s capability as a platform for *Virtual Enterprises* [FWA+98]. Major constraints that are discussed in this paper are those as evident in a closed hypermedia system like the Web.

The problems and enhancements proposed are summarized as follows:

First Class Links: Web lacks a rich linking mechanism that can efficiently maintain and manage highly complex and interdependent relationships between rich artifacts in a complex information system. The fact that the Web allows broken or *dangling links* shows it is focused more towards *distribution and scalability* than *relationship management* [2]. As a solution the authors discuss a hybrid approach of extending the HTTP protocol to introduce the flexibility of an open hypermedia system in the Web using *Link Servers* that provides compatibility between new and older applications based on HTTP.

Notification: The absence of a built-in event notification mechanism in the Web makes it very difficult to maintain changes and update relationships among resources and documents. Considering the possible bottlenecks of notification techniques like *periodic polling* the authors suggest using the *First Class Link* as a notification mechanism provided there exists a support for *remote link authoring.*

Client Architecture: When it comes to usability the client architecture plays a crucial role. As an end-user of a system directly interacts with the client application and the user interface, these need to be simple and intuitive. Few of the Web browsers have established themselves as the de facto client applications for the web. It is difficult and sometimes inappropriate when these browsers are to be used as client applications that need to render complex information structure and visualizations. Plug-in and wrapper mechanisms to extend a browser’s capability can help, but there still remains the necessity of the big monolithic browser to be active all the time. *Data Specific Handlers* can be introduced to handle the complexity of diverse data formats in a complex information system. Well defined *interface specification* on the other hand will ease the binding of different elements of a hypermedia workspace. This eventually will enhance *coordinated tool interaction and the hypermedia workspace.*

Distributed Authoring and Versioning: For any collaborative work to succeed all the participants must be able to access the repositories of artifacts and project documents in a controlled fashion. There must be a standard, real time file-system like interface to such repositories that addresses issues like *authentication and access control*, concurrent access control, *versioning* of artifacts, namespace management and *metadata* support. WebDAV, a web standard now, is suggested as a solution to this issue.

Co-ordination: All of the issues discussed earlier finally lead to a major problem of *coordination* in *virtual enterprises*. It is expected that incorporating all the upgrades to the Web as discussed in the paper will help getting a task *oriented view* of a project rather than a *data oriented view* thus enabling a smooth coordination in distributed projects on the Web.

Further Issues

There still remains work to be done to bring about a real life like interactivity among co-workers in the Web. As the browsers still are prevalent as the major web clients, alternative client tools and user interface mechanisms have to be discovered. Tool unification can be difficult when they are not easily extensible.

Real time communication/messaging and threaded discussion boards have been effective in enhancing the communication among project peers in the Web but it seems there still lacks a standard life-like approach of doing it.

The Web is based on the client-server architecture. The role of HTTP and the Web in true distributed environments based on peer-to-peer architectures that do not require a central storage and control might be a new area of exploration.

Conclusion

The overall picture the paper presents from a Software Engineering viewpoint is easy to understand and appreciate. Software development is definitely a collaborative effort. A generalization of solutions to the problems in Software Engineering can give a framework of solutions for other domains too. Complex information systems are good examples of Hypermedia systems as they constitute documents of numerous formats intricately related to each other. The paper discusses the issues related to the Web as a Hypermedia platform.

The key issue the paper addresses is still valid in today's context. Web is increasingly gaining its popularity as a place for collaborative work practices. Organizations today still might be bounded by proprietary tools and information systems. A standard and simple solution to the problem of tool integration for openness is surely a sign of success towards solving the problems prevalent in a highly distributed collaborative work space. Extending and reusing a dominant protocol like HTTP surely will help achieve goals in establishing successful *Virtual Enterprises* with less effort and much productivity.

References

[FWA+98] R. Fielding, E.J. Whitehead, K. Anderson, G. Bolcer, P. Oreizy, and R. Taylor. "Web-based Development of Complex Information Products". Communications of the ACM, August 1998, pp. 84-92.

[2] User Interfaces and Hyperware, http://www.ics.uci.edu/~taylor/ICS221/slides/hypermedia.pdf