Capturing Communication and Context in the Software Project Lifecycle

Vera Zaychik
Drexel University
3201 Chestnut Street
Philadelphia, PA USA
zaychik@drexel.edu

William C. Regli
Drexel University
3201 Chestnut Street
Philadelphia, PA USA
regli@drexel.edu

ABSTRACT

Email is a principle form of communication in the software development process and a rich source of information about project history and design rationale. Extracting the correlations between email conversations and design changes can be difficult without some knowledge of the context in which the conversations occurred.

To improve project communications among software engineers and create more structured email archives we created CodeLink. CodeLink integrates email-based collaboration with the software development process, providing teams of developers a means to automatically associate specific programatic features, functions or code elements with email messages. In this paper we describe CodeLink's architecture, user interface, and results of an informal user study. We believe that, by integrating collaborative work tools with development tools, we can enrich the communication among engineering teams and build repositories that detail collaborative decisions made in the development process. These repositories can then be used to improve software maintenance and extract design rationale.

1. INTRODUCTION

Project knowledge is contained within code, requirements and design documents, bug databases, communications between developers, and the memories of individual developers. Project communication and collaborative exchanges contain a great deal of knowledge about design intent, rationale and decision making that could be harvested to improve project management and software maintenance. However, it is very difficult to extract what is useful from informal communications because these mediums are often poorly structured.

Electronic means of communication between software engineers (e.g., email, instant messaging, cooperative work tools) have become widely used but they still lack support for certain important factors of software development process. For example, in open-source projects, where participation is open to the whole software engineering community and is not restricted to any geographic area, email is the dominant communication medium. In recent years the trend is toward increased use of asynchronous communication over face-to-face meetings: rather than walking to the next cubicle, a request for information does not have to be satisfied immediately by the recipient but it is placed at sender's convenience. While email tools have become very sophisticated tools for communication, they lack the ability to provide *context-based* support to the collaborative work process.

This paper presents an approach to enable context-aware email-based collaboration among software developers. In face to face communications there is a sense of context. Informally, context is a sum of information we readily obtain from the participants by paying attention to surroundings and non-verbal cues: the exact subject of conversation, turn taking, etc. It is the collection of circumstances or conditions in which the communication act occurs. When we want to specify something, we can often point. Such facilities are not available in email. In email, when a need arises to specify a point in a referenced document, the sender is often forced to describe the location (third paragraph, second line or class `Foo`, function `Bar`, line 154).

Informal communication contains a great deal of project-related information, often information not found anywhere else [26]. The more developers rely on electronic communications, more data can be available from the communications to be used in the future. Email archives of mailing lists and email discussions are often maintained by the organizations. Those archives exist in parallel to the software project files and documentation, providing insufficient search capabilities: the information might be there, but most people are not willing to sift through hundreds of messages for the relevant few that deal with a particular part of the project.

Our approach is to enable developers to automatically include *context hyperlinks* to specific places in a software project source file or document. This, in turn, allows us to approach the problem of loss of information: messages are archived based on the developers context at the time the messages are sent. Our research aims to study the following questions: (1) can we automatically extract useful project information from email communications between designers using minimal interference and avoiding text parsing and analysis, instead using context extraction at time of composition of the message? and (2) can we improve the com-

munication between the developers by providing tools that allow them to see the relationships between code elements and communications?

We developed a software tool called CodeLink, which enables inclusion of context references/links to specific parts of source code into an email messages. When a developer composes a project-related email, the source code files she or he is working on are analyzed to extract semantic information (e.g. what function is the developer referring to?) and a code snapshot is taken to be archived in a dedicated CVS[12] archive. All email messages containing code context links inserted in above-described manner are archived in a database and indexed using semantic information extracted. These communications can be annotated, linked to any URL and arbitrarily grouped.

We believe that this work introduces a novel way in integrating Computer Supported Cooperative Work (CSCW) tools (such as email) with the software development process. In this way, relationships between code changes and collaborative discussions about code can be created and used by developers and project managers to better understand the thinking that went into the creation of the system.

This paper is organized as follows: Section 2 provides an overview of related work, Section 3 describes CodeLink in detail, while Section 4 describes a user study we conducted. Section 5 concludes this paper by outlining limitations of the current approach as well as future research directions.

2. RELATED RESEARCH

We briefly describe related research in Computer Supported Cooperative Work (CSCW), Context Awareness, and Design Rationale that has influenced this project.

2.1 CSCW and Context Awareness

Email has become the primary message tool used by 97% of North American knowledge workers on a regular basis. It outpaced other media as the preferred way to receive or give input during work (66% vs. 13% for face-to-face meetings and 12% for phone) [1]. Mailing lists are widely used to coordinate open-source projects[11], but email has also become not uncommon when communicating with someone in the next office or even in the same office[15, 8]. At the same time, it still presents many problems when used for certain tasks as it is not adapted for handling process and social context[24], workflow and negotiation. For a comprehensive review of studies of asynchronous communication see [20]. The advantages and deficiencies of using email specifically in Software Design are described in [23].

One aspect of communication still not well supported by asynchronous tools is context awareness. In face-to-face interactions, a great deal of information is expressed by using cues, which help in grounding between participants[9]. These cues are usually termed *context*. Context availability can improve the search capabilities by enabling more precise queries, which can result in enhanced recall and precision of results. Thus it is important to obtain context of email communications at the time of their creation. Then search of archived messages can be made based not only on text parsing, but also on context.

Most context-aware applications have been developed in the domain of mobile and wearable computing, and in domain-independent groupware. There are some exceptions. For example, Anchored Conversations[7], an application-independent tool for collaborative authoring, provides a chat utility with anchors which act as substitutes for deixis. However, the notion of context is used very narrowly, only enough information is extracted about the environment to allow anchors to be placed unambiguously due to the domain-independent nature of the tool. While CodeLink is a domain-specific tool, in other respects Anchored Conversations is most closely related to this research.

Usually new tools have to be introduced to the users instead of the well-known common ones. This results in disruption of the design process and possibly in rejection of the tool. To deal with this problem, Grudin[14] suggests building on existing and accepted tools where possible. A custom email client that demonstrates a particular functionality is likely to not include most other features of Microsoft Outlook or Netscape Communicator. Another common problem is that for such tools to be successful they have to be adopted by all members of the group[14]. This is commonly referred to as *Critical Mass or Prisoner's Dilemma*. In this research we added functionality to a commonly used email system to increase the chances of adoption.

2.2 Design Rationale

Over the past several years software projects have become increasingly decentralized – we live in the age of global economy. Software projects grow too large to be handled locally, outsourcing development activities to remote locations. Combined with the high turn-over rates, the problem of coordination is becoming more and more complex. One aspect of coordination is how to preserve and communicate the *how* and *why* of the development, information very valuable for maintenance and evolution of the software. The term most widely used in the research literature to describe these concepts is *Design Rationale* (DR).

Documenting DR during the design process has been shown to be a vital approach to improved correctness and speed in both Engineering[3] and Software[2] domains. In the areas with use of mostly ad-hoc tools, such as open-source projects, the need is felt most strongly: although the mailing lists of communications between developers are archived and available online, the lack of structure creates a barrier to effective retrieval and management, and thus an entry barrier for new developers to join[11].

Although most DR systems to-date are either generic or tailored to solve engineering or architectural design problems, several systems have been developed specifically for Software Engineering. Comet[19], a commitment-based system for sensor-based tracker software, uses explicit representation and reasoning with commitments to aid the software development, especially when considering reuse or change of a certain module. Comet analyzes the source code to get commitments, structure and behavior specifications of modules, to perform impact analysis. Developers can also explicitly state commitments. COMANCHE (COoperative MAintenance Network Centered Hypertextual Environment)[5], a multi-user language-independent environment for cooperative maintenance, allows different programmers to concurrently access and manipulate information related to maintenance requests, the design and implementation decisions made, and their motivations. It allows programmers to annotate any form of textual documents to provide *Rationale in the Small*, that is rationale concerned with implementation activities (as opposed to Rationale in the Large, con-

cerned with design activities). The PPIS (Process and Product Information System)[21], an information and browsing system for software design and evolution, provides a general purpose hypertext environment. Designers place and move objects and can attach links and annotations to them.

Since design rationale research originally started with argumentation approach, most systems to-date rely on user intervention to gather information. This approach has met with limited success because it demands substantial designer time to enter information[6] or alters the design process[10]. Designers are reluctant to document their actions during the detailed design process[13], and there are significant difficulties in getting them to use argumentation schemas to structure their thinking during real design tasks[4]. Due to this, several automatic design rationale capture systems have been developed in the last 10 years.

There are two general approaches to automatic design rationale capture. The first approach is to create a system specialized for a certain domain with well-defined semantics and/or put certain constraints on the design process. Another general approach to automatic design rationale capture is based on the communications perspective. This perspective states that design discourse, i.e. naturally occurring communication among the group members in the process of design, contains the design rationale and it can be captured without user intervention by recording the thoughts rather than shaping them. A study of a design team involved in conceptual mechanical design by Yen at al[26] showed that formal reports accounted for only 5% of the total noun phrases, while hypermail archive (email) contained 43%. Noun phrase metric for engineering design has been introduced by Mabogunje[18] to access the design process and predict design team performance. A strong correlation was found between the success of a product as measured by expert evaluation and the number of distinct noun phrases found in documentation. The general disadvantage of automatic capture method is that recorded information lacks structure and is difficult to retrieve in a systematic and meaningful manner.

Most communication-based systems allow import of multimedia data and hyperlinks between the artifact and other data. The result is a web of information with links to requirements, deliberations, simulation and analysis results, etc. This is an electronic equivalent of a design notebook. The HOS system[16] provides an environment for computer network design with facilities to import email and news files. The structured rationale is supported through *incremental formalization* by using simple text analysis and domain knowledge. HOS makes suggestions for formalization to the user for possible addition of links within the acquired information. Notice: the burden of importing relevant information into the system stays with the designer, but once the information is inserted, it can be linked to other objects.

When the communication information captured is not structured in a formal way, but is rather a web of hyperlinked objects, then it is not a Design Rationale environment in a strict sense of the definition. Instead, it is a *Design History* environment. The difference is that in a design history software, explanations and answers to specific queries are not provided. Rather, the user has to look through the supporting documentation to find out the answers. The environment merely provides a convenient way to attach and later locate the relevant information. OzWeb[17], a hypercode environment for software development, uses WWW technology (HTTP and HTML) to provide access to source code and supporting documentation and allows incremental addition of links as useful connections are discovered.

The main disadvantage of systems that require user importing the data is that the effort required is too great with no clear short term value. The users have to perceive a clear benefit to using the system or the effort required has to be minimal[14]. We counter this problem by automatically capturing the email exchanges between the developers and providing the ability to include deictic references.

3. CODELINK ARCHITECTURE AND IMPLEMENTATION

CodeLink is based on the following observations:

1. Designers and software developers resent interruptions and resist process changes. Due to this manual design rationale capture methods have been generally unsuccessful in the industry. The goal is to capture process knowledge with minimum overhead and the least interference.

2. Automatic capture methods made some headway, but encountered the problem of lack of structure. Informal communications such as email exchanges are easy to capture, but difficult to retrieve efficiently. At the same time, we still want to capture informal communication as it proved to contain a great deal of process and product information.

3. Email applications are domain-independent tools. While this fact has lead to wide acceptance of email in the workplace, it also caused loss of context information in communication. Users find workarounds for this problem, but missing context can result in vagueness and misunderstanding. To insert context information manually some effort is required.

The goal of CodeLink is to automatically and unobtrusively extract the software development context that should be associated with email-based project collaboration. CodeLink uses these emails, along with code snapshots, to build a repository which can be searched in a variety of ways to improve the software development, management and maintenance process.

3.1 Approach

To access context information within the development environment we couple the email client with the software development environment. When a developer points to or is editing a certain piece of code, what information is relevant and important to the current communication? In our approach, there are several important pieces of information that are available for extraction. Not all of them are necessary for immediate goal of pointing out a piece of code to recipients of communication, but they become important for structuring and indexing accumulated communication data.

A message *context*, C, for an exchange between software engineers, is defined as a tuple $C =< P, T, E >$ at time t where: P is the project information about which the subjects are communicating, T is the task in which the author of the message is engaged at and shortly before the exchange,

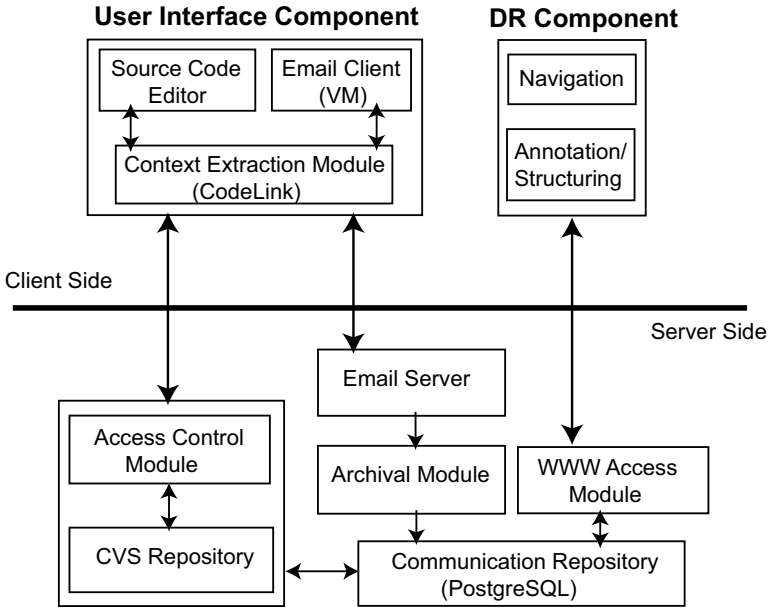

Figure 1: CodeLink Software Architecture.

and E is the personal environment of the author of the exchange. In turn, P can be defined on different levels of abstraction:

1. On the topmost level, P consists of project *name* and *location*. This can also include *package* name if available.
2. When the project is in the development stage (as opposed to design), the additional level is specific file/files information: *file name* and *version*.
3. In object-oriented programming the software is broken up in logical functional units called *classes*. In such case class name is part of the project information.
4. *Function name*. Functions are groups of statements performing a particular functionality.
5. *Line number*. The exchange can be on the level of a particular programming statement.

Of the above levels of abstraction of project information, our current implementation considers the following: file name, line number, enclosing function, enclosing class, enclosing package, CVS repository containing the file (project name), and CVS root (project location), i.e. the central location of the repository that is accessed by all developers. Version number is currently not extracted. If CVS version control is not used for the project, the name of the directory containing the file is assumed to be the name of the project.

While this schema originally supported only Java source files, most programming languages map to this schema in some way or other. Additionally, function, class and package names can take on 'not applicable' value. This is due to the fact that the line pointed out by the developer does not have to be part of any function, class or package. This is the case, for example, if the selected line is the import statement at the beginning of the file. In the same way, C++ code does not have a notion of packages and that value is always 'not applicable'. Figure 4 provides the list of languages and context-based cues supported in the current CodeLink.

This approach to context is in a sense very limited: T and E of the context C definition are currently not extracted, i.e. we ignore the task the user is concentrating on, his/her actions right before the exchange, other source files being modified. One could inquire as to the connection between the exchange and the recipients of the message. The email might be addressed to another employee on the same hierarchy level or to someone higher, a manager perhaps. This might have significance to the exchange and for later retrieval, but it is not easily extracted and even more difficult to analyze. Additionally, a piece of information that would be very useful but not easily available is the intent of exchange: is the message request for information, a reply, or perhaps a notification of change. This information can be somewhat reliably extracted using speech act theory or by requiring the author of the message to specify this explicitly. Extraction of these important pieces of information is part of our future research goals.

3.2 Software Architecture

The software architecture consists of several server and user modules, as shown in Figure 1.

- On the user side, a **context extractor** and **mime handler** are responsible for enabling sending and receiving messages with references.
- A web browser allows access to the **online browsing/search interface** to the communication database, and to the history of source files annotated with relevant messages.
- On the server side, several services enable archival of messages sent, storage of file snapshots and the interface to the database. All server-side components

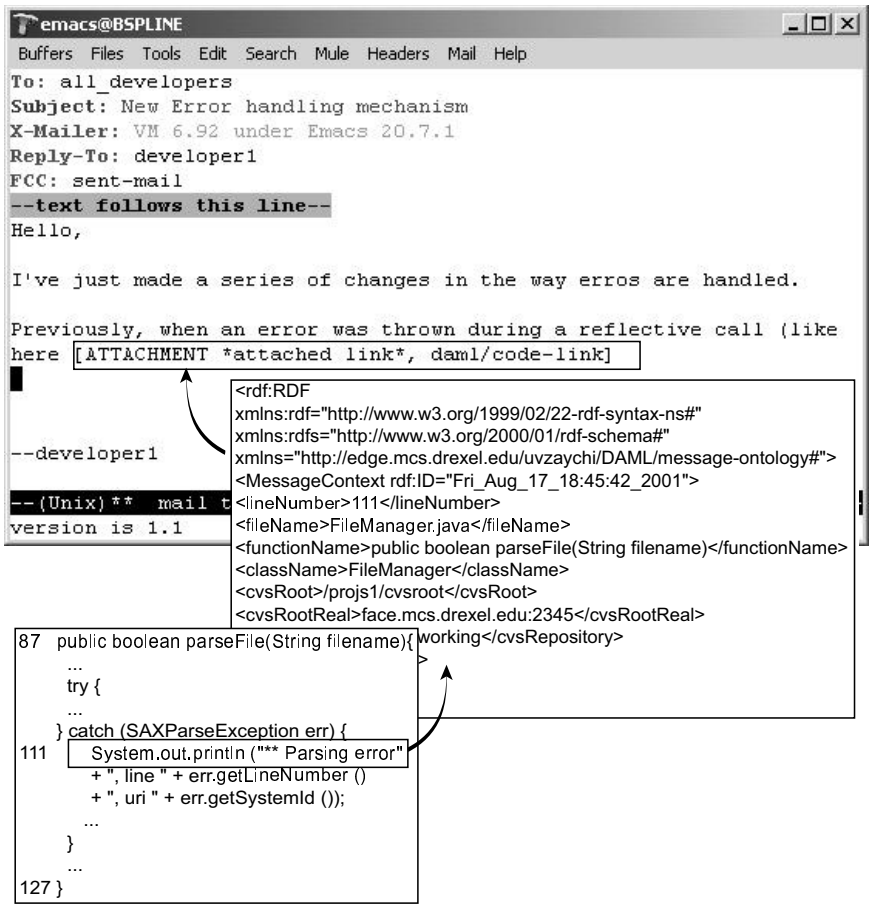

Figure 2: An example of context extraction during the context link inclusion process.

except for web interface are implemented by creating a separate Unix user. This user runs all services and is the owner of the CVS and PostgreSQL databases (PostgreSQL[22] is an open-source Object-Relational Database Management System).

The context extractor, when invoked by the user, analyzes the source code and extracts relevant information. This information is encoded using DAML ontology and inserted into the email message as a MIME attachment of type daml/code-link. At the same time, a snapshot of the source file is taken and sent to the CVS access control module using CVSPUT request. Thus only the context link is attached to the message, files themselves are not. See Figure 2 for an example of context extraction during the link inclusion process.

DAML is the DARPA Agent Markup Language, based on Resource Description Framework (RDF) and Extensible Markup Language(XML) and being developed by the W3C and Semantic Web communities. DAML provides a set of constructs to flexible describe knowledge, create ontologies and to markup information so that it is machine readable. A main motivation behind this language is to describe information contained in the Web pages so that computer agents can read and interpret it; however it can also be used as a knowledge representation language.

We use DAML for two primary reasons: first, its flexibility and extensibility. Our DAML ontologies for context and collaboration can be improved and refined, extended to other collaboration modalities (e.g., audio conferencing) and remain backward compatible with our initial definitions. Second, the DAML-based context descriptions are semantically grounded and suitable for use, downstream, by inference tools that extract design rationale patterns.

Any number of context references/links can be inserted in any particular email message. Any message containing references to code is automatically forwarded to the Archival Server. The user can also independently cc: any message he/she feels is important to archive to the server as it has a dedicated email address.

Mime Handler. Once the recipient receives the message, code references can be displayed using a special mime handler for daml/code-link attachments. This handler parses the DAML-encoded attachment and sends a CVSGET request to the server specified in the reference. It gets the file back and displays it as an HTML file with a bookmark to the sender's selection.

CVS Access Control Service. On the server side, a special CVS repository is set up for referenced code files. However, it is not accessed directly. Rather, a custom access control service receives CVSGET and CVSPUT requests and fulfills them. When fulfilling a CVSPUT request, the method of retrieval and the version number of the file are

returned. The file requested is returned for CVSGET requests. In both cases, if the request is faulty, error is returned to the client. Several client requests can be handled concurrently. The CVSGET requests are also received from the Web browsing/search interface.

The archival server receives all the email messages from the Email server. It parses the messages and the daml/code-link attachments and saves them to a PostgreSQL database (. The database has a web browsing/search interface. This interface allows browsing by author, date, project name, cvs root location, package, class and function names. The messages can also be grouped and browsed by groups. The user can add comments and context links to any message in the database. While browsing, the user can narrow the query if too many matches are returned by any of the other browsing criteria. See Figure 3 for an example of web interface browsing by project name.

History-annotated code. Another module annotates the source file with context links to the messages about the specific lines of code by interfacing to the message database. This module can be very useful since it maps messages back to the artifact presenting them in the original context while not requiring tight integration or storage in the artifact.

3.3 Implementation

The above described architecture has been implemented in a prototype called CodeLink currently running in our laboratory. For the prototype of the system we use Emacs as a source code development environment. Emacs is a very popular GNU software and is widely used in research community. Additionally, it provides a language called ELisp (Emacs Lisp) using which additional functionalities and modules can be developed and distributed by any Emacs user. Over the years Emacs has grown to include version control interfaces (for example, pcl-cvs is an emacs interface to CVS), inline web browsers (W3), email clients (VM) and many other extensions. We use VM[25] as an email client for this project because it runs inside Emacs and is written entirely in ELisp, which makes it very easy to interface with. VM is an open-source software, that has usual email client functionality, as well as more advanced commands that complete tasks like bursting and creating digests, message forwarding, organizing message presentation according to various criteria, and creating rule-based virtual folders. This also means that the module is system-independent, i.e. it can run equally well on Windows, Unix or any other platform as long as Emacs and VM are installed. Although VM is not the most widely used email client, it has the same attractive quality of Emacs - ease of integration and extensions. Other widely used clients, such as Microsoft Outlook or Netscape Communicator, have limited or difficult to work with interfaces.

The context extraction is written in ELisp and uses mode-specific Emacs functions. Currently, Java, C++, C and Perl contexts are supported, however a context link to any type of file can be inserted, in which case function, class and package information is not extracted. Table 4 demonstrates the mapping of context concepts for different languages. In Perl sub is mapped onto function name, and class name is always 'not applicable'. When linking to C source files, both class and package name are always 'not applicable'. After extracting language specific information about the selection, we need to find out whether the current file is a part of some CVS repository. For this, we look for directory named CVS in the parent directory of the file. If such a directory is found, the file Root states the location of CVS root, and the file Repository – the name of the repository/project. If such information is unavailable, the file is assumed to not be a part of any CVS repository, and the name of the parent directory is used instead. It is assumed that most groups do use some version control system and CVS is the system of choice for most open-source projects.

It is evident that once the file changes, the context link might no longer be correct. In order to deal with this problem, a copy of the current state of the file, snapshot of sorts, is saved and sent to the CVS server. This is achieved by making a direct connection to the custom access control service with a CVSPUT request. The service returns the version number of the snapshot and the exact method to get it in the future to be included in the context link. This way the correct version of the file is always displayed when using links. The latter is included in the link so that no one central CVS repository is required for all users. As long as the context link contains the method to get to the snapshot, it is not important what particular repository is used by one user or another. This implementation is not the only possible one, nor necessarily the best one. Currently, the access control service can become a bottleneck if a lot of requests are dispatched. Instead the files can be inserted as attachments directly to the email message. We decided against such implementation, however, to not overburden the email message with possibly many different attachments and also not to confuse the user.

The information thus extracted is encoded using DAML ontology, the current version of which is located at the GICL website[1]. The encoded information is included as a MIME attachment of new type daml, sub-type code-link. When the user execute a 'send' command on the email message with attachments of type daml/code-link, such message is automatically Bcc: to the archive. All server-side services except web interface are run by a special user on a Unix/Linux platform. This user has an account and an email address. To this email address all the messages are forwarded. In current implementation we have such a special user set up on a Linux server.

3.4 Scenarios

To illustrate the approach, here are two scenarios of use of CodeLink.

Scenario 1. A group of software engineers are working on a new software project. Developer 1 is working on error correction. Previously, the errors were handled using one Error class and strings. Developer 1 created several subclasses to handle different errors more specifically. He checks the code into the code repository the group is using for version control and sends an email out to all developers to let them know they need to update their code to this new error handling model. The developer wants to provide code details in his message, refer to specific parts of the implementation, and provide examples of use of the new model. Problem: there is no easy way to insert context links to specific code instances.

Scenario 2. Developer 1 is assigned a bug dealing with a certain functionality being unavailable in one of the modes of

[1] http://edge.mcs.drexel.edu/uvzaychi/DAML/message-ontology.daml

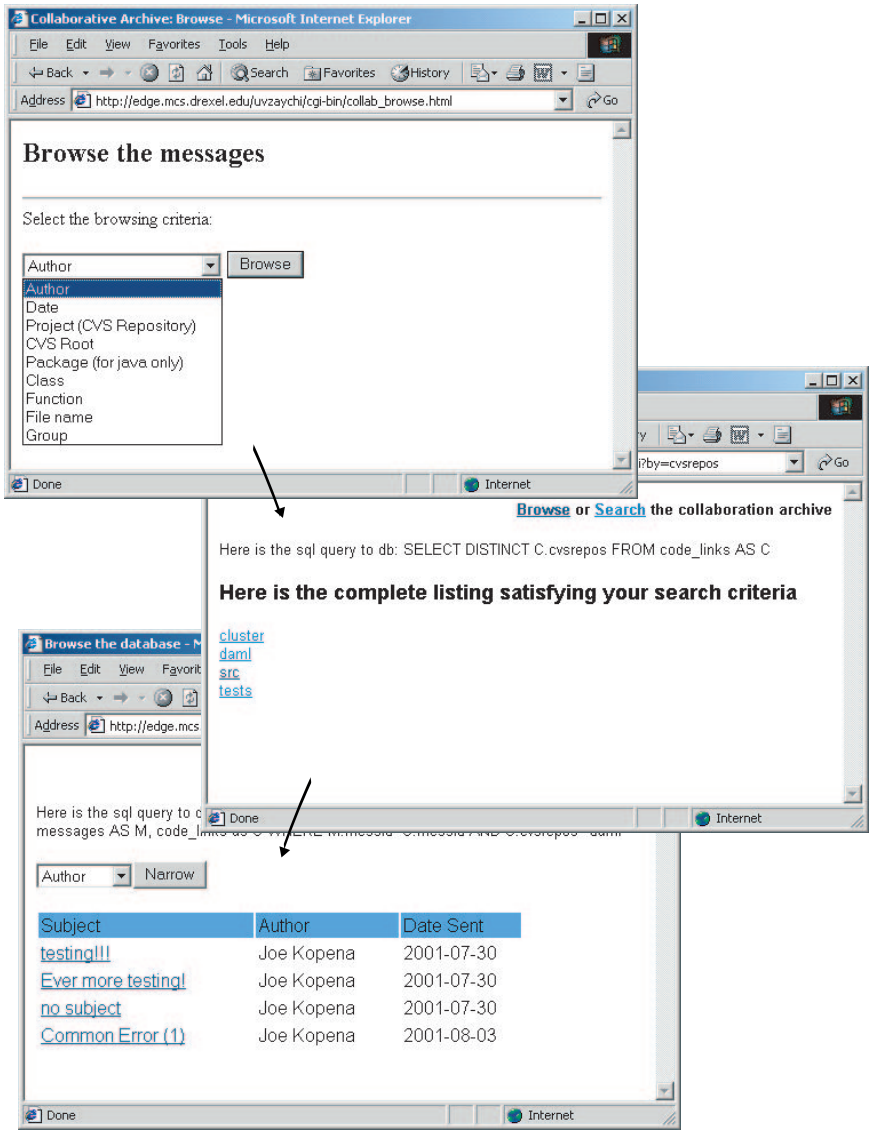

Figure 3: Example of web interface browsing by project name.

the software. He traces the code and discovers that the functionality in question is specifically disabled for that mode, but no reason is given in the comments. He removes the restriction but the resulting software produces incorrect results or crashes. It is evident that there was a reason for the original functionality, but where is this information contained? Problem: all changes to software have reasons behind them, but it is not easy to find such information after the fact.

In this work we present a systematic approach to deal with the problems in both of these scenarios. By solving the first problem we find a solution to the second. In Scenario 1 the problem that the developer is facing is lack of ability to express context information along with the content. Scenario 2 describes a more important problem of missing information. Using our approach and CodeLink, the scenarios can continue as following.

Scenario 1. Developer 1 inserts a context link to the old version of error handling code and explains why such model was not sufficient for the project. He does that by simply invoking a menu option in the email client and pointing to the buffer containing the code (Figure 5). He then inserts a context link to the new implementation of error correction using the same method and another link to an example of how errors should be handled from now on. Other developers on the team receive the email and are able to click on the links and see the change in the code. The file in question is opened in a browser and jumps directly to the selection made by the originator of the message. They also look at the example to make sure they understand the new approach. Developer 2 finds an inconsistency between the new code and the example and answers the original email pointing to the problem. She also has a question as to the overall effectiveness on the new approach. Developer 1 fixes the inconsistency and also answers the question of Developer 2. Other developers also have minor feedbacks about

Field	C++	C	Java	Perl
Function name	return_type *name* (args) {	return_type *name* (args) {	return_type *name* (args) [throws exception] {	sub *name* [(args)] {
Class name	class *name* {	not applicable	class *name* [extends, implements] {	not applicable
Package name	not applicable	not applicable	package *name*;	package *name*;

Figure 4: Context equivalents for different languages

the change and answer the original email. Every one of the above mentioned messages is forwarded to the archival database for maintenance purposes.

Scenario 2. Developer 1 retrieves the history of the source file in question and finds out who wrote the lines and when. He also discovers that several messages have been sent about the lines in question at the time of the original implementation (Figure 6). He reviews the messages and discovers that the functionality does not apply to this mode and would not make any sense. One of the messages also contains a context link to the white paper on the subject. He then removes the changes he made and also inserts the line of comment explaining the exception. The bug report is closed with detailed explanations and links to the messages in the archive.

4. USER STUDY

We conducted an informal user study to ascertain the usefulness and usability of CodeLink. CodeLink was made available to three groups of users (each group with 2-to-3 developers) for a period of several weeks. All three groups have been working on software projects for a long time before our software was introduced into their communication process. Users were instructed to only use CodeLink when the need for it is felt, and not in all project emails. Hence, the study was not designed to produce statistically significant conclusions, however several interesting patterns were observed and conclusions can be drawn.

1. Users found no or little problem in inserting or displaying context links. They chose to do so quite often. Most of the email messages with links were meant to point out a specific piece of code that needed work, had a bug, or that somebody needed help with.

2. CodeLink was extensively used in a guru-novice scenario, where one person knew the product or programming language to a much greater extent than the other. In such cases, the guru used context links to refer to specific parts of code for explanation, while the novice used them to ask questions. Email messages composed in such a scenario were very descriptive and contained a great deal of information about the software code. Such messages could be very useful for other novices on the project in the future.

3. Most user problems were caused by their unfamiliarity with the email client, VM. Only one developer was a novice Emacs user, and that person sent the least amount of messages. In short, people better acquainted with the email client and the development environment were more likely to compose messages with context links.

4. In several cases, users chose to manually `Cc:` the archival server on the project messages without links. They explained that they deemed those messages important for the record and forwarding to the archive manually was easy and fast.

5. CONCLUSIONS

This paper presented out work on CodeLink, a tool to integrate email-based cooperative work with the software development process. We believe that this work presents several research contributions. First, we introduced an approach to integrating generic CSCW tools with software engineering tools to enable extraction of domain-specific context for communications. This approach is very general and can be applied to other engineering domains and media other than email. Second, we informally showed that the ability to insert context links into messages can improve communications among engineers by reducing time required to specify references to code and removing ambiguity. Lastly, the archives of context-based email communications can be used to support the software management and maintenance process; enabling a wide range of queries, groupings and interrogations to find the important patterns and messages that occur during the project lifecycle.

CodeLink is not a complete solution. It does not have to be a stand-alone application. CodeLink can be a part of some bigger suite of tools, such as SourceForge. In fact, any system that has as its part a mailing list can benefit from integrating linking ability to code. Hypertext systems would benefit to an even greater extent because more pieces of information can be linked together.

There are limitations to our approach. First, the current prototype system is not very extensible, i.e. other context information pieces cannot be easily added or changed on the fly. Thus while the overall approach in this paper can work for many different domains, context extraction mechanism and the message archive have to be tailored to the chosen domain, in our case, software engineering. This problem might be solved by using more advanced database techniques. At the same time, the DAML ontology would need to be extended to allow for different domain contexts using inheritance mechanism.

Second, and more significantly, the current system is based on an assumption that developers send a great deal of messages with code references and that the ratio of such messages to all the project messages is high. If future user stud-

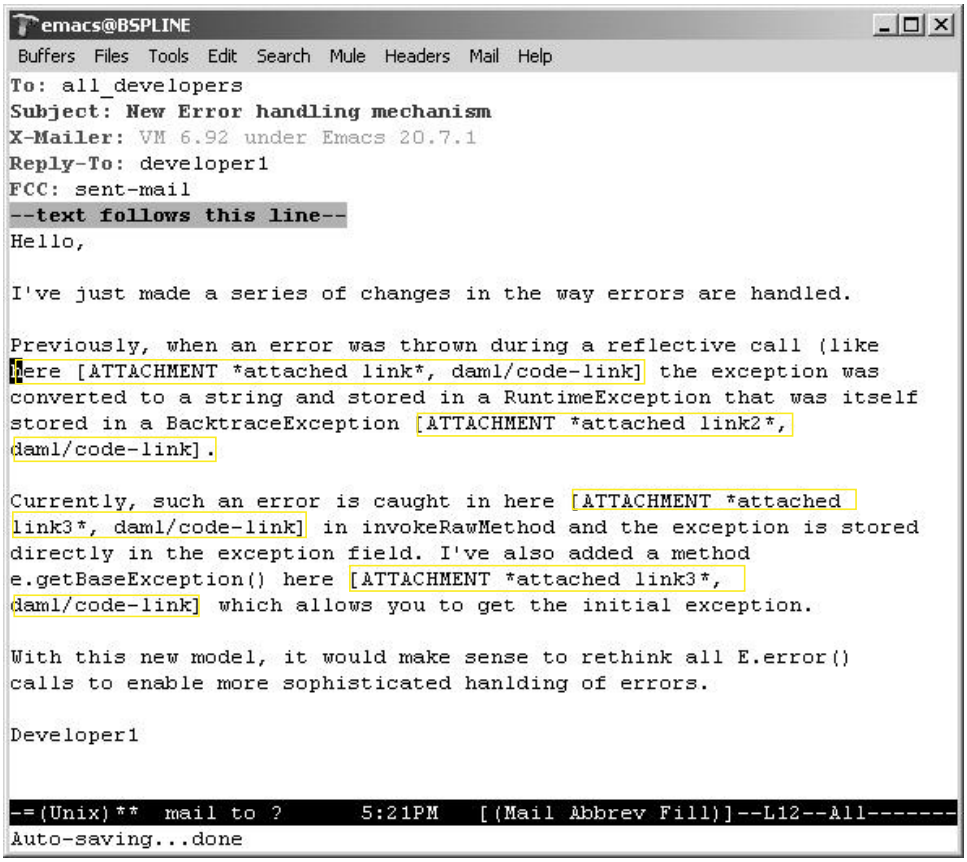

Figure 5: Screenshot of the email message composed by Developer 1 with context links inserted.

ies prove this assumption to be false, then the whole approach needs to be rethought. This ratio depends on many things: how distributed the work process is, what stage it is in, the roles between the developers. For example, in a mentor-trainee relationship there is a high likelihood of request-reply exchanges with code references in the replies. From our observations in open-source projects the size of the project and the number of the developers makes a difference in the ratio of code-related messages. Older and bigger projects usually draw a great deal of participants and the ratio becomes smaller. Overall, only numerous user studies can show how much relevant information can be captured with the approach described in this thesis.

Our current work does not consider security and privacy issues, and such matters can be very important. Developers can feel apprehensive about their messages being archived since such record might be used against them in the future (to show their incompetence, for example). There can also be a sense of 'big brother watching'. In the current approach not *all* messages are archived, but only the ones that are known to have project-related information due to the links to code. All other messages are ignored, whether they are personal or project-related. This results in a trade-off of getting more sense of security in using the application but losing project-related messages without links. This problem should be investigated in detail.

The security of company-owned information is also important. Currently, the message archive (and thus parts of the code) is available online for all the world to see. If CodeLink is to be used in a corporate environment, the access to the archive should be limited. One possible solution is to place it on the Intranet rather than Internet.

Our plans for future work on this project include several different directions: visualization, user studies, integration with other communication media, extending and deepening the notion of context, and natural language processing combined with different retrieval strategies for explicit extraction of design rationale.

6. ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation (NSF) Graduate Research Fellowship and Knowledge and Distributed Intelligence in the Information Age (KDI) Initiative Grant CISE/IIS-9873005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or the other supporting government and corporate organizations.

7. REFERENCES

[1] Messaging for innovation: Building the innovation infrastructure through messaging practices. A Study by Pitney Bowes, August 2000.

[2] L. Bratthall, E. Johansson, and B. Regnell. Is a design rationale vital when predicting change impact? - a

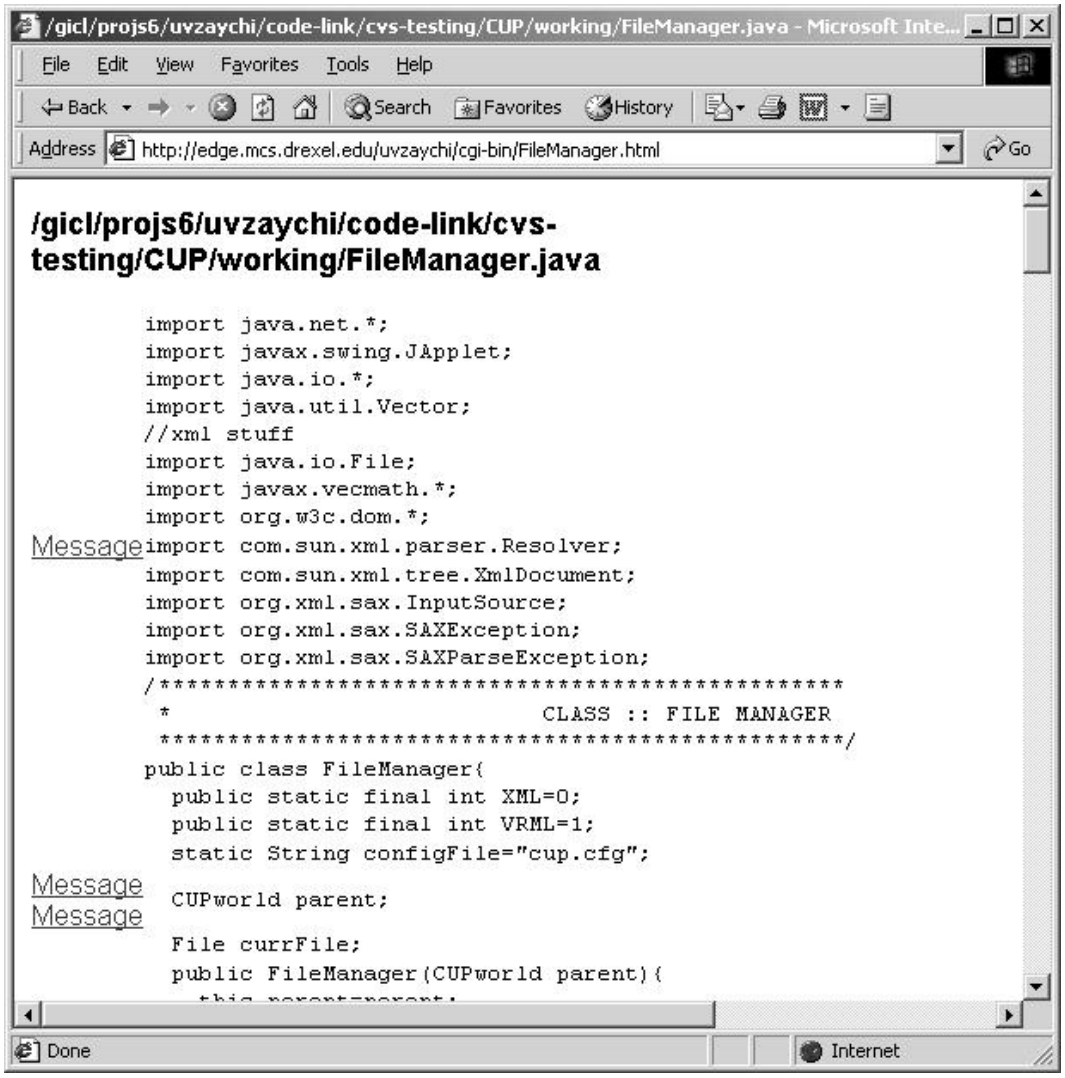

Figure 6: Screenshot of blame-annotated code with context links to relevant messages.

controlled experiment on software architecture evolution. In *Proceedings of PROFES'00: 2nd International Conference on Product Focused Software Process Improvement*, Oulu, Finland, June 20-22 2000.

[3] F. M. T. Brazier, P. H. G. Van Langen, and J. Treur. A compositional approach to modelling design rationale. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 11(2):125–139, April 1997.

[4] S. J. Buckingham-Shum and N. Hammond. Argumentation-based design rationale: What use at what cost? *Human-Computer Studies*, 40(4):603–652, April 1994.

[5] G. Canfora, G. Casazza, and A. D. Lucia. A design rationale based environment for cooperative maintenance. *International Journal of Software Engineering and Knowledge Engineering*, 10(5):627–645, October 2000.

[6] J. M. Carroll and T. P. Moran. Special issue on design rationale. *Human-Computer Interaction Journal*, 6(3-4):197–442, 1991.

[7] E. F. Churchill, J. Trevor, S. Bly, L. Nelson, and D. Cubranic. Anchored conversations: Chatting in the context of a document. In *Proceedings of the CHI 2000 Conference on Human Factors in Computing Systems*, pages 454–461, The Hague, The Netherlands, April 1-6 2000. ACM, ACM Press.

[8] V. Cicirello, A. Harris, E. Lee, V. Thomas, and V. Zaychik. A study of teamwork and collaboration in software design. Human-Computer Interaction Class Project Report, May 1999.

[9] H. H. Clark and S. E. Brennan. *Perspectives on Socially Shared Cognition*, chapter Grounding in communication, pages 127–149. American Psychological Society, Washington, D.C., 1991.

[10] J. E. Conklin and K. Burgess Yakemovic. A process-oriented approach to design rationale. *Human-Computer Interaction*, 6(3-4):357–391, 1991.

[11] D. Cubranic and K. S. Booth. Coordination in

open-source software development. In *Proceedings of the 8th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 61–66, Palo Alto, CA, June 16-18 1999. IEEE, IEEE Press.

[12] Concurrent versions system. http://www.cvshome.org.

[13] G. Fischer, A. C. Lemke, and R. McCall. Making argumentation serve design. *Human-Computer Interaction*, 6(3-4):393–419, 1991.

[14] J. Grudin. Groupware and social dynamics: Eight challenges for developers. *Communications of the ACM*, 37(1):92–105, January 1994.

[15] J. Hollan and S. Stornetta. Beyond being there. In *Proceedings of ACM Conference on Human Factors in Computing Systems(CHI'92)*, pages 119–125, Monterey, CA, May 1992. ACM, ACM Press.

[16] F. M. S. III and R. J. McCall. Integrating different perspectives on design rationale: Supporting the emergence of design rationale from design communication. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 11(2):141–154, April 1997.

[17] G. E. Kaiser, S. E. Dossick, W. Jiang, and J. J. Yang. An architecture for www-based hypercode environments. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 3–13, Boston, United States, May 17-23 1997. ACM, ACM Press.

[18] A. Mabogunje and L. J. Leifer. Noun phrases as surrogates for measuring early phases of the mechanical design process. In *Proceedings of the 9th International Conference on Design Theory and Methodology (ASME/DETC)*, Sacramento, CA, 1997. ASME.

[19] W. Mark, S. Tyler, J. McGuire, and J. Schlossberg. Commitment-based software development. *IEEE Transactions on Software Engineering*, 18(10):870–886, October 1992.

[20] J. E. McGrath and A. B. Hollingshead. *Groups Interacting with Technology: Ideas, Evidence, Issues, and an Agenda*, volume 194 of *Sage Library of Social Research*. Sage Publications, Thousand Oaks, CA, 1994.

[21] S. Monk, I. Sommerville, J. M. Pendaries, and B. Durin. Supporting design rationale for system evolution. In W. Schäfer and P. Botella, editors, *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, pages 307–323, Sitges, Spain, September 25-28 1995. Springer-Verlag.

[22] http://www.postgresql.org.

[23] M. Saeki, S. Sureerat, and K. Yoshida. Supporting distributed individual work in cooperative specification development. In S. Bhalla, editor, *Lecture Notes in Computer Science, Proceedings of the 6th International Conference on Information Systems and Management of Data (CISMOD'95)*, pages 232–247, Bombay, India, November 15-17 1995. Springer.

[24] K. Sproull. The nature of managerial attention. In L. Sproull and P. Larkey, editors, *Advances in Information Processing in Organizations*, pages 9–27. JAI Press, Greenwich, CT, 1984.

[25] http://www.wonderworks.com/vm.

[26] S. J. Yen, R. Fruchter, and L. Leifer. Facilitating tacit knowledge capture and reuse in conceptual design activities. In *ASME Design Engineering Techinical Conferences, 11th International Conference on Design Theory and Methodology*, New York, NY, USA, September 12-16, Las Vegas, NV 1999. ASME, ASME Press. DETC99/DTM-8781.