Weaving a Social Fabric into Existing Software

Li-Te Cheng, John Patterson, Steven L. Rohall, Susanne Hupfer, Steven Ross

IBM Research
Collaborative User Experience Group
Cambridge, Massachusetts

{li-te_cheng,john_patterson,steven_rohall,susanne_hupfer,steven_ross}@us.ibm.com

ABSTRACT

Contextual collaboration is a promising approach to embedding new collaborative features into existing applications. However, incorporating such new features may be too difficult for applications without extensible frameworks or too complex for legacy, custom, and mission-critical applications. We present Aspect-Oriented Retrofitting as a lightweight approach to embedding contextual collaboration in this class of applications, describe guidelines for designing retrofitting aspects, and walk through two examples.

Categories and Subject Descriptors

D.2.13 [Software Engineering]: Reusable Software – reuse models; H.5.3 [Information Interfaces and Presentation]: Group and Organization Interfaces – Computer-supported cooperative work, collaborative computing

General Terms

Design, Human Factors, Languages, Theory.

Keywords

Computer supported cooperative work, groupware, aspect oriented programming, application retrofitting, software reuse.

1. INTRODUCTION

Software features that enable users to communicate and collaborate with others are becoming more prevalent. Applications such as email are built entirely around such "collaborative features," but some applications that have traditionally lacked collaborative features have begun to incorporate them as well. Examples include games, programming environments with built-in software configuration management, and various office productivity applications.

This approach is known as "contextual collaboration," because it embeds collaboration into the context in which users work. Collaborative features manifest themselves as components within the running application and use the details of the context to enrich the collaboration. Contextual collaboration lets users work together without leaving their core applications.

AOSD 05 Chicago Illinois USA

Some applications – such as those with extensible frameworks that allow incorporation of new components at runtime -- are amenable to contextual collaboration. Others may require a developer to extend objects provided by an application programming interface and then rebuild. Another group of applications is even less amenable to contextual collaboration, requiring the developer to change the core codebase and rebuild the system. This group includes legacy systems, in-house/custom-built software, and mission critical applications. For these applications, it may be too expensive, complex, time-consuming, and risky to add a few new collaborative features.

We contend that contextual collaboration should be possible in any existing software application, not just those with extensible frameworks. A "retrofitting" approach is needed for non-extensible applications or software too onerous to change directly. In this paper, we present the use of Aspect-Oriented Programming (AOP) [13] as a means of retrofitting contextual collaboration, with minimal impact, on the host application. We first elaborate the motivation for contextual collaboration. We then examine various retrofitting strategies before describing why AOP lends itself well to retrofitting. Next, we present design considerations from contextual collaboration that aspect-oriented retrofitting should take into account. We then explain two examples illustrating these design considerations and conclude by discussing the consequences of our approach and how AOP can be further extended to support contextual collaboration.

2. CONTEXTUAL COLLABORATION

In this section we elaborate the notion of contextual collaboration. We provide a definition and describe the concept's key benefits and motivators [11].

2.1 Definition

People who want to collaborate with one another may use general-purpose collaborative systems incorporating a set of multi-user tools (e.g. shared editors, chat utilities, whiteboards), or they may use the specialized, single-user tools of their trade (e.g. spreadsheets, CAD, IDEs) and co-opt existing communication tools such as email. In the former approach, collaborators need to "go there" – to the team room or workplace – to work together on shared artifacts. In the latter approach, people "stay here" in their conventional tools -- leaving them in order to do limited, ad hoc collaboration -- and the artifacts end up scattered among participants' email inboxes, file systems, and other tools.

Contextual collaboration is a promising third approach that brings the collaboration "into context". Users are not forced to leave their core applications to launch collaborative tools or to visit a distinct collaboration platform; instead, collaborative

capabilities are simply available as components that extend standard applications [9]. People continue to use their customary applications, in order to collaborate with their colleagues about the project at hand.

2.2 Benefits

Perhaps the most significant benefit of contextual collaboration is that it can *reduce friction* [4]. By embedding collaboration seamlessly into core applications, users are spared the time and effort of context switching to other tools whenever they need to work together and can stay focused on the task at hand.. One example of contextual collaboration is the "Live Names" feature in IBM Lotus Workplace. Names appearing anywhere in this system (e.g. in an email) can serve as a launch point for collaboration, such as indicating online status, and initiating a chat [24].

A second benefit is that context can be used to *enhance collaborative work*. For example, consider the ad hoc collaboration that happens when workers communicate through email or chat applications about a document they are constructing together. If the conversation represents a particularly useful exchange of information, a participant is likely to archive the email or save a chat transcript. But consider what happens when one later wants to retrieve the discussion. Was it in email or in a chat? Who saved it, and where? Even if transcripts and emails can be located and examined, the work they reference may not be obvious, because the discussions are completely decoupled from the work artifacts. Churchill et al.'s tool allows text-based chats to be "anchored into" the documents that are the basis of the work [7]. These contextual chats are accessible to participants from icons appearing in the document's text -- allowing users to easily locate and revisit discussions.

Contextual collaboration can also better *inform collaborative work*. Consider what happens when a user initiates an anchored chat: All participants immediately know what work is being discussed; there is no need to tell them where to navigate or to paste in relevant text. If co-workers are using core applications that have been outfitted with contextual collaboration, those applications will have knowledge about each user's current actions – such as editing a certain file, debugging code, or chatting with co-workers – and can furnish that information. Better awareness of colleagues' context can forestall duplication of effort, inform whether to interrupt someone or not, and so on.

2.3 Motivators

Grudin offers two challenges for developers of collaborative applications that are strong motivators for retrofitting contextual collaboration into software [10].

The first is "unobtrusive accessibility", i.e. not designing for infrequently used features. Grudin advocates adding collaborative features to an already successful application rather than building a new one. Retrofitting fits well in this case. The collaborative features being retrofitted into the application are secondary to the original, heavily used features, and should appear and operate under the host application's design principles and avoid overshadowing the primary features.

The second challenge is "managing acceptance", i.e. winning user adoption. Retrofitting an accepted application with collaborative features sidesteps the problem, since the user is already largely familiar with only seemingly minor changes, rather than a brand new application that may require nontrivial retraining and reconfiguration.

3. RETROFITTING COLLABORATION

Contextual collaboration provides compelling opportunities to retrofit new collaborative features into existing applications. In this section, we discuss the challenge of retrofitting and describe past approaches.

3.1 The Retrofitting Challenge

The problem of retrofitting a set of collaborative features into an existing application is related to the general problem of software customization. Mørch breaks this problem into three levels: customization, integration, and extension [19]. Customization involves configuring the application through user-defined settings. Integration involves incorporating new functionality within the application's capabilities without accessing the underlying source code (e.g. macros). Extension allows radical changes in the application, completely unanticipated by the application's designers, through the introduction of new code. We consider retrofitting contextual collaboration as extensions to an existing application.

A lesson that we can apply to retrofitting from customization and integration is that we should minimize the impact of changes on the application. This lesson ties with Grudin's challenges (section 2.3): new collaborative features should play by the application's rules, and not place overhead on users. However, in the case of the software process of retrofitting, the "users" include the developers responsible for maintaining the application.

Therefore, the challenge of retrofitting contextual collaboration is to produce an extension to the host application that minimizes any changes to the original codebase and its build process. The extension -- consisting of new code implementing collaborative features -- should strive to operate within the host application's design principles.

3.2 Past Approaches

There have been examples of retrofitting collaborative features into existing systems from the literature of Computer Supported Cooperative Work (CSCW). These examples can be grouped at three levels – the application level, the programming environment level, and the operating system level. A set of desirable characteristics for retrofitting can be drawn from each of these levels.

3.2.1 Application Level

Retrofitting at the application level enables the developer to leverage any extensibility offered by the application's architecture. The chief benefit is that any collaborative features that are introduced will exist gracefully within the application. Ideally, the framework for extension would focus on the application-specific issues and insulate the developer from low-level details outside the application.

Examples include using APIs intended for third-parties to hook in new components (e.g. Churchill et al. use Microsoft ActiveX application interfaces to anchor chats inside Word [7]), or creating a proxy service to intercept and change the application's protocols for communication and presentation. However, the

original application architects cannot be expected to foresee every future contingency, and the available application programming interfaces and standard protocols may be limited or nonexistent.

3.2.2 Programming Environment Level

Retrofitting can also be considered at the programming environment level: one may be able to exploit the runtime characteristics of the environment used to create the application. Some programming language environments are flexible and offer options for programs to modify themselves at runtime and dynamically load new modules, without requiring recompilation. The main benefit here is the potential to significantly customize the application's behavior beyond the original design.

Programming language environments may be flexible, and offer options for programs to modify themselves at runtime and dynamically load new modules. Other environments offer some flexibility in manipulating the language's runtime libraries for UIs and event handling, without rebuilding the entire application. For example, through a custom class loader, Flexible JAMM replaces Java's single-user interface components with collaborative equivalents at runtime [3].

A problem with this approach is that not all programming environments have the needed flexibility. The application being retrofitted may be coded in a restrictive environment, or have requirements for strict control over runtime configuration that may deny modification access to runtime libraries. Also, if well-defined APIs are not available, it may be difficult to customize or introduce new behaviors. For example, while it might be easy to replace the default label widget with a new one by replacing the widget library at runtime, specifying that only one particular label use the customized widget might not be possible this way.

3.2.3 Operating System Level

The final level to consider involves diving into the operating system to trap event calls, capture screen pixels, and hook into the boundary between the application and operating system services. A significant advantage of this option is the ability to treat the application like a "black box". This is especially useful for old applications whose documentation and source code may be lost. Another consequence of this "application independence" is that techniques used to retrofit one application may work for another. Many application-sharing systems take this approach (e.g. Microsoft NetMeeting [17]), enabling them to share entire desktop applications.

There are some drawbacks to the operating system level approach. While the application becomes a "black box," the developer must now focus on the operating system's intricacies. The deep semantics and application's data structures are also obscured; only events and visible UI elements are discernable. Moreover, there may be interference from events from other services running in the operating system.

3.2.4 Three Desirable Characteristics for Retrofitting

Each level highlights a diverse array of examples and suggests desirable characteristics to help retrofitting. The application level spotlights access to the application's deep semantics through clearly defined programming interfaces. The programming level points out the flexibility afforded by modifying runtime configurations. The operating system level showcases the richness of trapping events.

4. ASPECT-ORIENTED RETROFITTING

Aspect-Oriented Programming [13], or AOP for short, is a desirable approach for retrofitting. We refer to our use of AOP for retrofitting as Aspect-Oriented Retrofitting. We first provide some background about AOP and how it supports effective retrofitting. Then we identify issues and related work.

4.1 Aspect-Oriented Programming

Object-oriented programming languages help modularize software functionality into classes and methods. AOP is a methodology that extends object-oriented programming languages by providing constructs to help express concerns that span several classes and methods, which are difficult to express in an object-oriented hierarchical class/method framework [13].

A major benefit of this approach is the separation and modularization of secondary, supporting functionality (expressed as *aspects*), from the application's core objects. The aspects use a *join point model* to bind aspects to objects in the application at some level of granularity. With the aspects peeled away, the core objects are purely focused on core logic, not secondary logic.

Our discussion of AOP for retrofitting focuses largely on our experiences with AspectJ [1], a popular AOP extension to the Java programming language. However, the constructs we discuss may translate to other implementations.

4.2 Aspects Trap Event and Application Flow

The first characteristic of AOP that helps retrofitting is the join point model. Join points can specify points of runtime execution of a program. Join points can refer to a variety of operations depending on the AOP implementation, such as method calls, calls to constructors, attribute assignment, etc. Actions can be defined in an aspect to execute before, in place of, during, or after specific join points or groups of join points.

Thus a retrofitting aspect can express join points to trap key events in the flow of an application. Actions can then be defined to introduce collaborative features at appropriate points in the application.

4.3 Aspects Expose Deep Semantics and Are Part of the Application

The join points and actions expressed in aspects can be defined to capture valuable details from the running application, such as returned objects, exceptions, method parameters, and the calling object for a method. The aspect's actions can even invoke methods from captured objects. Unlike an external application restricted to monitoring events at the operating system level, aspects can expose the everyday constructs used internally by an application. Thus, a retrofitting aspect can capture details needed to provide the context for contextual collaboration, and access the internal API exposed by captured objects.

Also, the retrofitting aspect is a first-class object within the application's codebase. It can access the objects, methods, and fields from captured context and can execute actions around join points. Thus, contextual features introduced by the aspect operate under the same conditions as any other application feature.

However, setting up join points implies a priori knowledge and raises other issues, which are discussed in section 4.5.

4.4 Aspects Minimize Impact

There are two ways to introduce aspects into an application (termed *weaving*). The first approach is to use an aspect compiler that compiles the aspect and generates intermediate code or binaries that express the aspect in the application's original language through a variety of techniques, including reflection and event hooking. In the case of AspectJ and various other AOP implementations, the process does not require recompilation of the non-aspect code, only linking. The second approach uses a special runtime that dynamically incorporates the aspect code at runtime, which also requires no recompilation of the non-aspect code [2].

Thus a retrofitting aspect minimizes impact on the host application's codebase. No changes pollute the original codebase, and no recompilation of the original application is required. Only linkable binaries are needed, so even legacy applications without original source code are eligible for retrofitting.

Compiling and linking in the aspect does affect the build process, but impact may be minimal. The first approach to building an aspect only requires compilation of the aspect, generation of intermediaries that do not affect the original application, and linking. The second approach does not involve building or linking at all, but requires deployment of a special runtime to dynamically bind the aspect with the application.

4.5 Drawbacks

Despite its advantages, Aspect-Oriented Retrofitting has a number of potential drawbacks. These are: finding the right join points, expressing join points, restrictions of the programming environment, and overhead.

4.5.1 Finding the Right Join Point May Be Hard

A key difference between designing a retrofitting aspect and a regular aspect is finding the right join point. Traditionally aspects are developed along with the rest of the application and design documents and source code are readily available. Also, the developer can change non-aspect code to accommodate aspects (e.g. remove crosscutting method calls that will be replaced by an aspect). Under these conditions, a join point can be found in the source code, or the code can be changed to provide the right join point.

However, a retrofitting aspect faces a more restrictive design condition. The application source code and documentation might not be available. They might have been lost over time, or legal, political, and security issues may block access to code. The developer may have to resort to reverse engineering techniques, but some applications may be too complex to analyze.

Also, the retrofitting process seeks to minimize change on the codebase and the build process. Thus, the code should not be changed simply to facilitate a join point.

These conditions, which will vary from situation to situation, can restrict the available join points a retrofitting aspect can choose. A restricted set will limit how deeply contextual features can be embedded into the host application.

4.5.2 Join Points May Be Brittle

The difficulty of finding desirable join points means trade-offs must be made with the restricted set of join points. This highlights another difference between retrofitting aspects and regular aspects: join points in retrofitting aspects may be very narrowly focused on opportunistic calls that are vulnerable to change.

Given a more limited choice of join points, a retrofitting aspect may have to rely on coincidental calls and events in the host application to incorporate new features, rather than staying true to the application's semantics. For example, suppose a retrofitting aspect wants to introduce some initialization after the host application completes its own, but that the host application put all its initialization in the "main" block. The only available join point is when the application finally instantiates the UI after everything is initialized. Thus, the retrofitting aspect uses the join point and defines an action to execute new initialization routines before the UI is instantiated. Although this gets the job done, it takes advantage of the coincidence that the UI gets instantiated after the application's initialization. If the next major revision of the application removes the initialization, then the retrofitting aspect will fail and must find another appropriate join point.

4.5.3 The Environment May Not Allow Aspect-Oriented Retrofitting

Using AOP to retrofit is similar to retrofitting approaches operating at the programming environment level. The environment building and running the aspect is leveraging hooks, events, and dynamic loading capabilities from the programming language. Thus, the drawbacks of the programming environment level may apply as well. The original application's language may not support AOP extensions. The security model may prevent aspects from being incorporated with the application.

4.5.4 The Retrofitting Aspect May Incur Too Much Runtime Overhead

Although using aspects may eliminate the need to recompile the original codebase, and minimize the impact on the build process, the AOP implementation may incur memory or performance overhead on the host application. Using an aspect compiler to generate intermediate objects may add too many objects and increase the memory requirements. Using a special runtime to dynamically introduce aspects may slow down the application.

This is a problem with adding new modules to any system, however. Also, the additional overhead may be acceptable depending on the retrofit's requirements. Finally, AOP technology is constantly improving to address these overhead issues [21].

4.6 Related Work

There are numerous examples of building AOP applications that separate out secondary infrastructure capabilities related to collaboration, such as object persistence and authentication (e.g. see Laddad for detailed examples [14]), but not in the context of retrofitting new collaborative capabilities into the UIs of existing applications.

There is some work describing the use of AOP to design new collaborative systems and UIs. Veit and Herrmann extend the AOP paradigm with their own programming model to realize the

model-view-controller architectural pattern for building UIs [23]. Cardone et al. introduce new language constructs to decompose UI libraries by feature-encapsulating components to support multiple platforms from the same codebase [5]. While these approaches are valuable in architecting new applications or redesigning UI libraries, they do not directly address the problem of retrofitting new collaborative capabilities into old applications.

Mørch introduces a notion of aspect-oriented software components, which he uses to architect a flexible application for customization and runtime extension [18]. Mørch's approach, however, is still focused on architecting first for aspects rather than tackling a completely foreign, non-extensible application.

We have also conducted an investigation in using aspects to retrofit a collaborative capability (which will be summarized in section 6), but focused on UI-derived join points [6]. In this paper we examine a wider spectrum of potential join points that a retrofitting aspect needs to consider before introducing new collaborative features.

5. SOCIAL FABRIC: DESIGN GUIDELINES FOR RETROFITTING IN COLLABORATION WITH ASPECTS

Given the advantages and drawbacks of retrofitting aspects, now we need to consider how to design them for embedding contextual collaboration. This raises three questions. First, what kinds of concerns in contextual collaboration should retrofitting aspects address? Second, where are the relevant join points in the application's architecture? Third, what if the architecture lacks the relevant join points?

To answer these design questions, we propose some guidelines structured around the set of concerns that need to be introduced by the retrofitting aspect (which we call *social concerns*), and a set of layers describing the existing application architecture as well as the desired architecture. We call this overall set of social concerns and architectural layers the "social fabric": the missing pieces that the retrofitting aspect must add to realize contextual collaboration and the hooks in the host application that contextual collaboration will attach to.

5.1 Social Concerns

Our social concerns for contextual collaboration are inspired by consideration of the simplest social interaction: a conversation. Before a conversation can begin one must be aware of the availability of someone to talk to. We refer to this as social awareness. We refer to the actual conversation as a social interaction. As the conversation proceeds there are rules and expectations for how it will happen. These are referred to as social roles. We expand on these concerns in the sections that follow.

5.1.1 Social Awareness

Social awareness is a critical feature of collaboration. When we work with other people we are aware of their comings and goings, their availability for interruption, and their level of distraction. When we work in the same location, this knowledge of the context of our collaborative partners is not something we must seek out -- it's available for free if we bother to notice. This social awareness is the "backplane" from which our social interactions are launched.

When we are not in the same location, supporting social awareness is more difficult. Buddy lists are an attempt to introduce social awareness into a distributed work environment. Depending on our concerns for privacy and our familiarity with our potential collaborators, we may be willing to share much more information about our context, provided it requires no effort. Close collaborators might be welcome to know what applications we are running or which files we are working on. The trick is to make the information available without disturbing the person who provides it and without overwhelming the person who will notice it.

When we retrofit an application to support social awareness we are either exporting context from the application or importing context from somewhere else. Thus, a retrofitting aspect may need to implement two specific tasks:

- Collecting context that is offered for others to perceive,
- Presenting the context of others.

To collect context, the retrofitting aspect can specify join points to capture parameters, returned objects, calling objects, and other details. These details from the application should derive what can be offered to other users to perceive. An approach to consider for collecting context may be the "wormhole pattern" which allows context to be passed directly between points in the call stack [14].

> **Example**: Consider the case of retrofitting the ability to share an editor with multiple users. The act of loading a file and the filename can be captured at a join point associated when the application loads a file. This information can then be sent to other users to indicate that someone is loading a specific file in the editor.

To present context to others, the retrofitting aspect must transmit the contextual information across the network and eventually manifest it in the UI of another application.

> **Example**: In the shared editor example, the retrofitting aspect could establish a socket session with the other user's shared editor and broadcast the file loading event and the filename. Then the recipient aspect can display this information in the other editor's UI.

5.1.2 Social Interaction

Social interaction is what collaboration is all about. It can be done in many ways, but it is a dialog or conversation among two or more people about something. Sometimes the interaction is verbal, but it might also involve drawings or pictures. Sometimes the interaction is immediate as in a chat, but other times it takes place over time with no two participants working on the conversation at the same time.

When we retrofit an application to support social interaction we must both establish a channel for the conversation and provide the content that the users want to discuss. Thus, a retrofitting aspect will need to be concerned with:

- Providing a mode of interaction,
- Providing the referent of the interaction.

The mode of interaction has implications for the networking requirements of the collaborative feature being added and how the interaction will occur in the UI. As a result, the retrofitting aspect will need to specify join points and actions to set up networking and set up a UI for interaction. The nature of the interaction will

also affect how the network notifies the UI of new collaboration-related events, and how the UI will send collaboration-related information to other applications.

> **Example:** Consider the case of retrofitting a chat component into a spreadsheet application. The retrofitting aspect may choose to use the join point where the spreadsheet UI is being initialized, after that, the aspect initializes networking and the chat component's UI.
>
> Because chat messages can arrive at any time, the networking code must listen for incoming messages from the chat server. When a new chat message arrives, the retrofitting aspect must pop up a chat window. The window's constructor will need the spreadsheet's window object as a parent and thus it must be captured by a join point where that window instance is available. When the user finishes typing a reply in the chat window, the retrofitting aspect must then relay the reply back to the recipient through appropriate networking calls.

The referent of the interaction will vary with the type of collaboration (e.g. communication support, information sharing, workflow/coordination) and is not always needed. As in the *social awareness* concern mentioned earlier, our desire to provide a referent for the interaction is an effort to give it a context. In *social awareness*, however, we are only providing snippets of context -- enough to inform the decision to interact or not, but not enough to invade privacy or overwhelm the user's attention. To support social interaction, we now want major pieces of an application to be shared in a detailed manner as the backdrop or referent of the conversation. This retrofitting aspect will need to bring elements of the application into the communication channel.

> **Example**: In the case of the chat example, a retrofitting aspect might obtain context from a spreadsheet being edited (e.g. cell locations, data, formulas, etc) and bring it into the chat for discussion.

5.1.3 Social Roles

Social roles loosely define the script by which people interact. Normally, we simply "know" what's going on, but computers offer the opportunity to keep track of people's roles and even enforce them. This reification of social roles can be essential for blocking unwanted or inappropriate actions, but it can also be useful for informing users about what is expected of them.

When we retrofit an application to support social roles we introduce policy requirements and coordination rules. As a result, a retrofitting aspect will need to consider these two capabilities:

- Providing policies regarding user rights,
- Providing indications of user responsibilities.

Rights imply issues such as authentication and access control. A retrofitting aspect may be able to leverage frameworks available in the application, utilize services available in the organization, or come up with a custom-built solution. The aspect may need to identify join points in the application that need to validate incoming accesses from remote users. Access control and security are common examples in the AOP literature – see Laddad for examples [14].

> **Example:** Consider the case of retrofitting a shared task component into a personal calendar application, where task objects can be linked to calendar entries. The retrofitting aspect could use the corporate directory service to authenticate users. Users associated with a task object may want to retrieve the description in the linked calendar entry, so the join point around the method retrieving the calendar entry is needed, and the aspect must verify that only authorized users can obtain the entry's description.

Responsibilities imply issues around coordination and social norms. The retrofitting aspect will need to provide a UI that could assign responsibilities to users, or let users choose their own. The resulting system could be "laissez-faire", where nothing is expected, or very structured based on activities, workflow, and deadlines. Responsibilities also imply actions that the user needs to perform in the application, and a retrofitting aspect can help by identifying appropriate join points where these actions can be performed and any useful context.

> **Example:** In the case of the task assignment, the retrofitting aspect provides a user interface for assigning tasks to other users, and uses calendar details (e.g. dates and times) to define deadlines. When a user is assigned a task, the retrofitting aspect uses join points within the calendar object to display who assigned the task and how much time is left to complete it, as well as a button on the calendar entry widget to indicate completion.

5.2 Architectural Layers

Retrofitting contextual collaboration requires join points tying into the architecture of the application. Also it involves introducing new pieces of architecture that did not exist before, especially if the application was originally intended for standalone use. Regardless of whether they exist or not in the application, there are three layers to consider: the distributed system, the application model, and the UI.

5.2.1 Distributed System

The distributed system layer refers to how the retrofitted application communicates with external applications and services. It can refer to calls to the operating system, a traditional client/server networking model, or a peer-to-peer networking system. There are three cases worth considering:

- The layer is present and it can be used for contextual collaboration.
- The layer is present but it is inadequate for contextual collaboration.
- There is no layer or its join points are completely inaccessible.

The first case is very convenient, but probably specific to applications that already have some limited collaborative capabilities. In this situation, a retrofitting aspect only needs to find the appropriate join points to attach calls into the distributed system layer.

> **Example:** Consider the case of retrofitting a document-sharing component for a medical imaging application that uses a proprietary networking protocol to exchange images among users. Each time an image is sent, a

document should be attached to it. A retrofitting aspect can leverage join points into the module managing the proprietary networking to share documents in addition to images.

The second case is probably more common, and here the retrofitting aspect must introduce its own networking capabilities. There might be some useful context worth capturing from the existing networking layer, so join points into the existing layer may be needed.

> **Example:** Regarding the medical imaging example, suppose there is a join point to signal that an image was sent, but there is no way to add in documents with the outgoing image. A retrofitting aspect will need to define its own scheme to exchange documents (e.g. HTTP). It can still use the join point to capture the identifier for the image sent out, and execute calls using the custom networking scheme to share documents associated with the image.

The third case is also probably commonplace. Again, the retrofitting aspect must introduce its own networking capabilities. However, the join points will need to target places in the application model layer, or, failing that, the UI layer.

> **Example:** Regarding the medical imaging example, suppose there are no join points available around the networking module. However, there is a join point where the user clicks on the "send image" button widget. The retrofitting aspect can target this join point to share documents with its own networking scheme.

Retrofitting aspects operating at the distributed system layer may benefit from using distributed AOP implementations such as JAC [20].

5.2.2 Application Model

The application model layer describes the core logic of the application. An application may lack a distributed layer or a UI layer, but it will always have some sort of application model layer. Retrofitting aspects have the potential of significantly changing the behavior of the application by leveraging join points at this layer. Collaborative features that involve sharing application state as opposed to pixels or UI widget events (e.g. shared editors, shared debuggers, shared web browsers) benefit from accessing context information at this layer.

> **Example:** Regarding the medical imaging example, suppose that the application has a core object with a method called "sendImage(Image)". This is an ideal join point for a retrofitting aspect to capture image information, and implement document sharing.

5.2.3 User Interface

The UI layer provides the screen presentation and processes the user's direct input. If the user interface follows the Model-View-Controller paradigm, then this layer contains the View and Controller. The Application Model layer contains the Model. There are three cases to consider:

- The layer is present and is adequate for contextual collaboration
- The layer is present but it is inadequate for contextual collaboration
- There is no layer (e.g. the application is a system service), or there are no accessible join points to the UI

The first case is ideal, since the retrofitting aspect can introduce user interfaces that appear like natural extensions to the application. Retrofitting aspects can consider join points at the boundary between the UI and application model layers, as well as the boundary around the UI library itself. Veit and Herrmann's extensions to Model-View-Controller may help [23].

> **Example:** Regarding the medical imaging example, suppose the user receiving an image wants to read the associated document. A retrofitting aspect can find the join point where the image viewer window's menu is being defined, retrieve the menu widget, and add in an option to pop open the document in a text widget.

In the second case, the retrofitting aspect can still leverage join points where the user interacts with the UI. However, the retrofitting aspect must now use its own custom user interface widget to supplement the application's user interface. Cardone's mixin widgets [5] and JAC's UI aspects [20] may be helpful.

> **Example:** Returning to the previous example, suppose the document is in HTML, but the application's text widget does not support HTML. The retrofitting aspect can still use the same join points as before, but must use a custom widget to display the document.

In the third case, there are two choices. The retrofitting aspect can advise join points at the distributed system layer to pass application information to another application providing a UI, thus turning the main application into a server, and the UI application into a client. Another option is to add in a new UI layer by building on top of join points at the application model layer.

Each approach has its own uses depending on the requirements of the retrofit. If a clean separation between application logic and UI is desired, and the application is already capable of running as a server, then the first choice may be worthwhile. If the application is incapable of becoming a server, then adding in a new UI layer may be useful.

6. AN ADDRESSBOOK EXAMPLE

To explore aspect-oriented retrofitting for contextual collaboration, we began with a simple example focusing largely on the UI layer [6]. Here, we summarize the steps we took for the retrofit and reflect on how it ties into our notion of a social fabric.

6.1 From Address Book to Buddy List

We started with a basic address program, written in Java and using the SWT widget library [8]. The program is a single-user application that lets the user enter contact information, save and load all contact data, and conduct searches (top of Figure 1).

The address book's list of names was modified to present online awareness information provided by an instant messaging service. Our final result appears at the bottom of Figure 1. We have the same application, but now names are decorated with icons denoting online status. Tooltips over the names reveal status

messages. We did not have to recompile the original application and only added one aspect written in about one hundred lines of code that interacted with the IBM Lotus Sametime instant messaging toolkit [12].

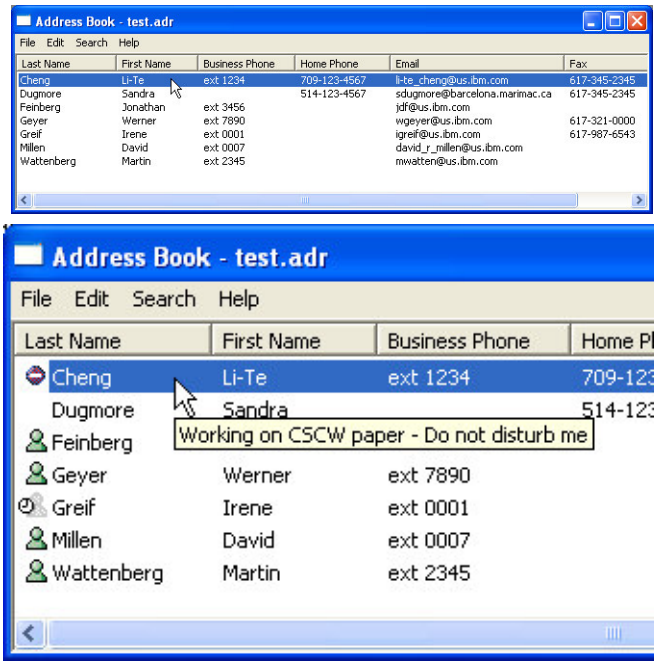

Figure 1: Original address book application (top) and the same application retrofitted with IM presence icons and tooltips (buttom)

6.2 Implementation

Our strategy to accomplish the retrofit was threefold. First: understand the application from its runtime behavior and its codebase, looking for useful internal application programming interfaces. Second: identify the join point and define the associated code where we can initialize the new collaborative feature upon application startup. Third: identify the join point and define code where we can establish a foothold into the UI and add to it.

6.2.1 Understanding the Application

From understanding the operation of the address book application, we learned that the address book is represented by an AddressBook class, which includes an open() method that is called when the application is starting up, and returns a Shell object (the widget for the entire application window). Thus, we can leverage the application model to help start up our new collaborative feature.

Also, the address book uses a Table widget consisting of TableItem widgets for each row. Each TableItem contains the fields for one contact, which are set up using a setText() method. The key field is the email address which we can use as an identifier to get online status information from the instant messaging service. There are also unused methods to set icons and tooltips in the table.

We then used AspectJ [1] to define an aspect representing instant messaging awareness information associated with a row entry in the table of the address book. Figure 2 shows the aspect we created, with the internals written in pseudo-code for brevity.

```
public aspect LiveName
{
  after() returning(Shell shell):                          A
    call(* AddressBook.open(..))                           B
  {
        // Display login dialog in shell
        // Login to IM System
        // Add listener for IM status changes, update icons
  }

  after(TableItem item):                                   C
        (target(item) &&
         call(* TableItem.setText(..)))                    D
  {
        // Get email from table item
        // Get current IM status using email
        // Get icon based on IM status
        // Add listener for mouse hover, show IM status text
        // Add icon to table item
  }
}
```

Figure 2: Pseudo-code showing the aspect responsible for retrofitting the Address Book application

after their respective join points.

6.2.2 Retrofitting Initialization

Section A defines a join point on any call to the open() method of the AddressBook class. This captures the moment when the application is starting up. This is an important moment to allow us to perform setup related to our new collaborative feature.

Section B captures the shell widget returned by the open() method and performs various actions after the application starts up. We pop up a login dialog using the shell widget as a parent object. After obtaining login information from the dialog, we log into the instant messaging service and set up a status change listener, responsible for updating icons in the table rows, using a hashtable that maps email addresses to rows.

6.2.3 Retrofitting the Existing User Interface

Section C defines a join point on any call to the setText() method of the TableItem widget, and captures the actual TableItem widget instance calling the setText() method. This specifies the moment when a row with contact information is being created or changed. This is an important moment to set up an awareness icon in the table, and establish tooltip information.

Section D takes the TableItem widget captured by the join point and extracts the email field from "item", which is then used to query the instant messaging system for status information. The status information is then mapped to appropriate tooltip and icon information to display in the table ("item" allows us to access the appropriate methods). The hashtable used in section B is updated with a mapping between the email address and the table row.

6.3 Social Concerns

Social Awareness (collect and present context): This example was about introducing social awareness into an application. The

context for the social awareness is provided by a separate application (a buddy list) and must be presented to the user. To do this we need to collect two pieces of information from the table of contacts: the appropriate contact field that we can use to query the instant messaging service for online status information and updates, and the table item widget to let us insert awareness icons and tooltips. If we can obtain the table item widget for each row, then we can display awareness icons and tooltips.

Social Interaction (mode and referent of interaction): In this example, there is no real interaction going on, since we are only displaying awareness information. However, the example could be extended to launch a chat (the mode of interaction) from the name. Moreover, the retrofitting aspect could bring in contact information from the address book as a context (the referent of interaction) for the chat.

Social Roles (rights and responsibilities): In this example, we need to log on to the instant messaging service in order to access online status information. This relates to Rights. There are no Responsibilities, since there are no expectations associated with monitoring online status and maintaining an address book. However, the example could be extended with an expectation that users should keep their contact information up-to-date. This could be implemented by reminding the user to verify contact information and automatically share updates with other users.

6.4 Architectural Layers

Distributed System: The address book is a standalone application. Thus, we introduced a networking layer via the instant messaging toolkit.

Application Model: Given that our collaborative features are largely UI-based (adding icons and tooltips), we only used the application model to trap program startup. For instance, we do not really need to manipulate contact information, and all of our context can be gleaned from the table widget.

User Interface: We needed a join point where the main window appears so we can pop up a login screen for the instant messaging service, establish our connection, and set up a listener for online status updates. Also, we needed a join point where a row in the table is updated or added. This is where we can insert our awareness icons and tooltips.

7. REPLICATED APPLICATION SHARING

A more complex example of aspect-oriented retrofitting is our Zipper project, which is still work in progress [22]. The goal of this project is to provide synchronous collaboration using distributed copies of single-user applications. After a brief introduction to the problem of replicated application sharing, we describe how it relates to our social fabric model.

7.1 What Is Replicated Sharing?

Collaborative use of replicated, single-user applications has long been a dream of groupware developers. If such a system were available, then the myriad of single-user applications could be repurposed as collaborative tools. Not only would people be able to collaborate, they would be able to collaborate with the applications to which they are accustomed.

On the face of it, the idea is simple. If all of the collaborators have a copy of the single-user application, then one user can

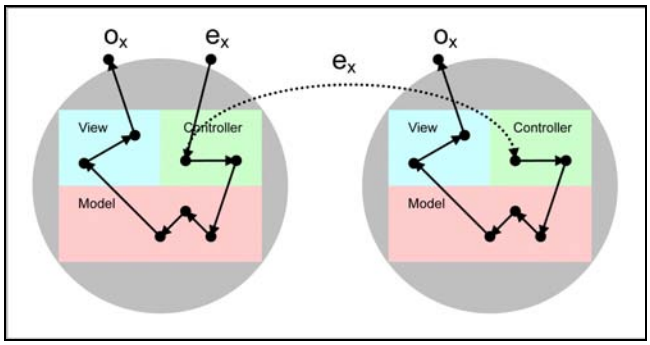

Figure 3: The replicated application sharing vision

"drive" all the application replicas. Underlying this idea is the notion that, if the same sequence of events (e.g., user input) is sent to replicas of the application, then the application state will be manipulated and modified in the same manner in each of the application copies—each collaborator will see the same result. (This can be seen graphically in Figure 3.) This is much more network-efficient than "screen scraping" systems such as NetMeeting [17], since the bandwidth of the input events is small compared to the application output that gets displayed to the user.

Figure 3 shows two collaborators (although theoretically there could be any number of collaborators); the collaborator on the left is the moderator interacting with the application while the collaborator on the right is observing the moderator's actions. When the moderator interacts with the application, an input event, labeled e_x, is sent to the controller for interpretation. Application logic will cause some sequence of operations as a result. This processing flow is shown as the black lines. The processing will wind through the model and the view and eventually produce some output, o_x. In replicated application sharing, the initial event, e_x, is caught and transmitted to the other collaborators (shown as the dotted line in Figure 3). The event is then interpreted by the remote collaborators. Assuming that the applications were in the same initial state, the same processing flow occurs and the same output, o_x, is produced.

The problem with this simple vision is that it doesn't work; it assumes state changes within the application are *deterministic*. In replicated application sharing, that assumption is usually incorrect. Whenever the application logic consults something in the environment outside of the application, such as a local file access or a system call to retrieve the time, the environmental access may return a different result. Subsequent processing may follow a different path through the two applications. Begole, *et al.*, call these environmental problems *externalities*. More specifically, they define an externality as an input (other than the user) or an output (other than the display) that is external to the application itself [3].

The Flexible JAMM system took one approach to fixing the problem of externalities. It exploited properties of the Java language to dynamically replace single-user components with specially-written multi-user counterparts. It did this to ensure that all the replicated applications experience a common environment by directing all environmental accesses to a common proxy server

[3]. Our Zipper system is investigating a different approach using aspect-oriented programming.

In the discussion that follows, we will concentrate on the aspect retrofitting in Zipper. Other issues of making a replicated application sharing system are described in [22].

7.2 Zipper Implementation

The Zipper system is designed to share editors within the Eclipse environment. Initially, we are targeting text-based editors with a goal of sharing other Eclipse editors. Sharing editors (for Java code, XML files, etc.) is one piece of a larger story on collaborative application development. Our goal is to share pieces of the Eclipse environment without needing to change any existing code, either in Eclipse or in Java itself.

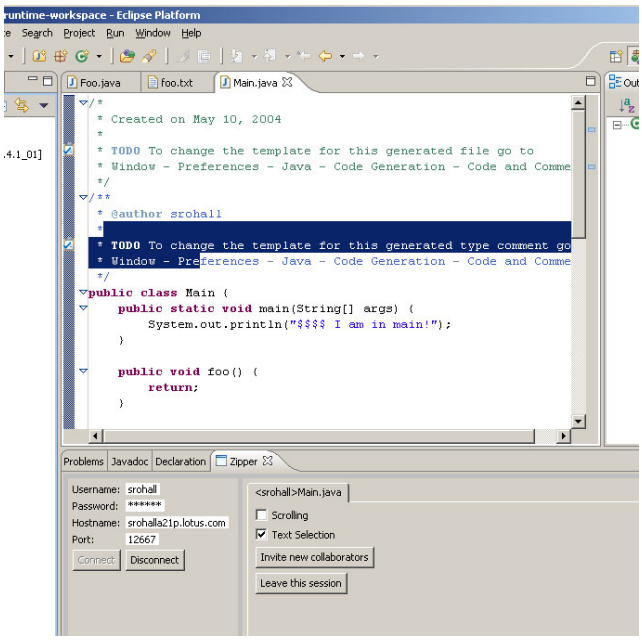

Figure 4: A Zipper-enabled Java editor in Eclipse. The sharing panel at the bottom allows text scrolling and selection events to be selectively shared; text entry events are always shared.

Figure 4 shows the current Zipper prototype. From the user's perspective, the Java editor, the main portion of the Eclipse screen, looks and behaves like any other Java editor in Eclipse. What is different is that a remote collaborator can make changes to the document. The Zipper view, below the editor, allows any collaborator to control the amount of sharing—in this case, text scrolling and selection events can be selectively shared with remote participants. Text entry or deletion events are always shared among collaborators.

We originally anticipated the use of two types of aspects. The first would involve aspects to catch the moderator's input events and share them with other users. The second would involve catching the environmental accesses on one replica and forwarding them to the other replicas ensuring that all of the applications experience the local environment in a common fashion. In this manner, we hoped to use aspects to preserve the deterministic assumption and ensure that the processing on all replicas is identical.

7.2.1 Capturing Input Events

As we have implemented Zipper, we have found that our use of aspects has changed from our anticipated uses. In particular, we found that Eclipse's UI widgets provide very rich interfaces for intercepting user input events. For example, we wanted to be able to share text selection events among users. As it happens, Eclipse's ITextViewer object defines an interface for registering selection listeners. Similarly, there are listener APIs for capturing text entry and scrolling events. Our problem stemmed from the fact that the ITextViewer objects to which we wanted to listen were hidden from us—ITextViewer objects for editing text are created as a by-product of instantiating an IEditorPart object—the actual UI widget used to contain the ITextViewer within the Eclipse framework.

Figure 5 shows an aspect we used to make up for this deficiency in the Eclipse API. It looks for the ITextViewer objects created as a result of an IEditorPart's UI widget being created. Once we have a handle to the ITextViewer being shared, capturing input events is simply a matter of registering the appropriate listeners with that object.

```
// Adds a new mapping of an IEditorPart to an ITextViewer.
after(IEditorPart e, ITextViewer v) returning :
  cflowbelow(
    execution(* IEditorPart.createPartControl(..))
    && target(e))
  && (initialization(ITextViewer.new(..))
      && target(v))
{
   editorToViewer.put(e, v);
}
```

Figure 5: Code showing the aspect responsible for finding the Eclipse ITextViewer object associated with an IEditorPart.

A related issue is interjecting events programmatically into the editor's processing stream. Again, in the case of textual editing, Eclipse provides the necessary APIs for creating events that appear as if they were the result of user interaction. Although Eclipse has been a good environment in which to catch and create these sorts of user input events, we can imagine that in other environments, or even for certain events within the Eclipse environment, we may need to use aspects rather than rely on listener APIs.

7.2.2 Capturing Environment Accesses

Unlike user input events, Eclipse does not have an existing API that can be used for capturing environment accesses. Instead, much of the environment of a file in an editor is provided by IResource objects. As we move forward, we will need to use additional aspects in Zipper to detect when the Eclipse workbench consults the IResource associated with a given piece of text being edited.

7.2.3 Code "Archaeology"

One unanticipated use of aspects in the Zipper project has been for reverse engineering Eclipse to determine which events need to be intercepted. In the Eclipse SWT graphics package, we can capture individual key down and up events. Alternatively, in the

Eclipse JFace package, those low-level keyboard events get aggregated into higher-level text events comprising a range of characters in the text to be replaced and the new replacement text. These higher-level events are more useful for replicated application sharing.

The use of aspects proved invaluable for determining which objects and events were most useful for collaboration. Most of these aspects were simple, logging aspects to help decipher the flow of control within Eclipse. Once the appropriate objects were discovered, these aspects were no longer needed.

7.3 Social Concerns

Social Awareness (collect and present context): Zipper is not directly concerned with social awareness. We expect it will launch from a separate system, such as an instant messaging client. As such it is a specialized conferencing tool that may be added to an ongoing chat. It would be useful if that client were augmented with information about which applications are Zipper-enabled and available to each of the participants in the chat.

Social Interaction (mode and referent of interaction): The purpose of Zipper is to permit any application to become the referent of a conversation. A chat or telephone call provides the primary means for direct interaction. Zipper provides the information about which to talk.

In addition to providing the context for a conversation, Zipper can also support some interaction through the application. If the user is the moderator, then he is driving the application just as he would if it were not being shared. A retrofitting aspect will intercept the moderator's actions and transmit it to the other collaborators where, once interpreted, the results are seen.

If a collaborator is not the moderator, then his interactions are limited to affecting only the surface presentation of the application in the form of a telepointer. Most replicated applications add a telepointer feature allowing the remote collaborators to point to things even if they are not currently the moderator.

Social Roles (rights and responsibilities): The primary roles in a replicated application are the moderator and the other collaborators. The moderator is allowed to interact fully with the application while the other collaborators are limited to viewing and interacting with telepointers. These roles are needed to ensure system synchronization, however; it is best if the collaborators do not need to be aware explicitly of the roles. The idea is to let the person who interacts with the system be the moderator. When there is no activity, another user can begin interaction with the application. At that point, they become the moderator without an explicit passing of control. In this case the retrofitting aspect needs to hook user input to the replicated applications and, based upon whether someone else is currently interacting with the system, allow the user to become the moderator or not.

7.4 Architectural Layers

Distributed System: The point of Zipper is to make single-user applications collaborative. As a result, these applications typically do not have a communication layer already. Zipper must add networking support for transmitting events among the collaborators.

Application Model: There are critical points of the application model which must be monitored in a replicated situation. Zipper must monitor any calls made outside of the application into the larger computing environment. An example is making a system call to get the current time. These accesses are externalities. Since the operating environments are not identical across collaborators' computers, the externalities on the moderator's computer must be tracked so that the same result can be sent to the other collaborators. Without doing this, it is very likely that the distributed applications will not maintain synchronization. (This problem is described more fully in [22])

User Interface: For typical interactions with the application, we need a join point wherever a user can interact with the system. For the moderator, this is so we can transmit the events to the others; for the others, it is so we can transmit telepointer information.

8. FUTURE DIRECTIONS

Contextual collaboration is an opportunity for existing applications to incorporate new collaborative features. AOP can extend the reach of contextual collaboration to applications without extensible frameworks. The main benefits of using AOP for retrofitting are: the ability to trap events and application flow, access to application semantics and context, and minimal impact on the original application's code base and build process. We also present the notion of social fabric to help guide the design of aspects that retrofit contextual collaboration into applications, and examples showcasing the potential of this approach.

There are drawbacks to Aspect-Oriented Retrofitting: finding the right join points, brittle join points, restrictions in the programming environment, and runtime overhead. Automated analysis of the event flow of applications [15], may help address the first and second problems. Work needs to be done in trust models for weaving in retrofitting aspects, and enhancing aspect runtime technology [21].

The potential for brittle join points has long-term consequences for maintainability. Over time, numerous retrofitting aspects may be applied on a complicated legacy system, which may make it even harder to understand. Thus, retrofitting aspects could be viewed as a tactical short-term patch for a long-term problem. This may suggest explorations of how to design retrofitting aspects to fit within a long-term strategy of migration towards a new system.

There is also an opportunity to extend AOP with new join point models capturing human-computer and human-to-human interaction, which will help better express join points in retrofitting aspects. These join point models would go beyond single UI events, and represent event flows in a UI model and workflows requiring multiple users to participate synchronously and asynchronously. The challenge of better expressing join points is related to the goals of Naturalistic Programming [16].

We are only speculating on the potential of contextual collaboration with aspects. Future research should implement contextual collaboration on large, complex, existing systems with Aspect-Oriented Retrofitting, and compare the results with other approaches. Such a comparison should consider not only the software engineering metrics, but also feedback regarding the user experience. Case studies can also inform our model of the social

fabric, and build a grounded set of guidelines to help future retrofitting endeavors. Finally, the hidden pieces of context and application interfaces uncovered by retrofitting aspects may lead to new creative and insightful uses of contextual collaboration. Through aspects, existing software applications -- as old and peculiar as they may be -- can still remain comfortable and familiar, while entering the new world of social software.

9. ACKNOWLEDGEMENTS

We would like to thank Martin Lippert for making his AspectJ-Enabled Eclipse Runtime (AJEER), which was used in the Zipper prototype, available (see http://www.martinlippert.com/eclipse-aspectj-runtime/index.html for more information).

10. REFERENCES

[1] AspectJ. http://www.eclipse.org/aspectj.

[2] AspectWerkz. http://aspectwerkz.codehaus.org.

[3] Begole, J., Rosson, M., Shaffer, C. Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems. *ACM Transactions on Computer-Human Interaction*, 6, 2 (June 1999), 95-132.

[4] Booch, G., Brown, A. Collaborative Development Environments, in *Advances in Computers Vol. 59*, Academic Press, August 2003.

[5] Cardone, R., Brown, A., McDirmid, S., Lin, C. Using Mixins to Build Flexible Widgets. In *Proceedings of AOSD 02* (Enschede, The Netherlands, April 22-26). ACM, New York, NY, 2002, 76-85.

[6] Cheng, L., Rohall, S., Patterson, J., Ross, S., Hupfer, S. Retrofitting Collaboration into UIs using Aspects. In *Proceedings of CSCW 04* (Chicago, USA, Nov. 6-10). ACM, New York, NY, 2004.

[7] Churchill, E., Trevor, J., Bly, S., Nelson, L., Cubranic, D. Anchored Conversations: Chatting in the Context of a Document. In *Proceedings of CHI 00* (The Hague, Netherlands, April 1-6). ACM, New York, NY, 2000, 454-461.

[8] Eclipse.org. Eclipse 2.1.2 Example Plug-ins, http://fullmoon.torolab.ibm.com/downloads/drops/R-2.1.2-200311030802/

[9] Fontana, J. Collaborative Software Ages Slowly. In *Network World Fusion*, January 6, 2003.

[10] Grudin, J. Groupware and Social Dynamics: Eight Challenges for Developers. *Communications of the ACM*, 37, 1 (Jan. 1994), 92-105.

[11] Hupfer, S, Cheng, L., Ross, S., Patterson, J. Introducing Contextual Collaboration into an Application Development Environment. In *Proceedings of CSCW 04* (Chicago, USA, Nov. 6-10). ACM, New York, NY, 2004.

[12] IBM. Lotus Instant Messaging and Web Conferencing, Sametime Java Toookit, http://www-136.ibm.com/developerworks/lotus/products/instantmessaging

[13] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. Aspect-Oriented Programming. In *Proceedings of ECOOP 97*. Springer-Verlag LNCS n. 1241, Germany, 1997, 220-242.

[14] Laddad, R. *AspectJ in Action*. Manning, Greenwich, CT, 2003.

[15] Li, D., Li, R. Transparent Sharing and Interoperation of Heterogeneous Single-User Applications. In *Proceedings of CSCW 02* (New Orleans, USA, Nov. 16-20). ACM, New York, NY, 2002, 246-255.

[16] Lopes, C., Dourish, P., Lorenz, D., Lieberherr, K. Beyond AOP: Toward Naturalistic Programming. In *Companion of OOPSLA 03* (Anaheim, USA, Oct. 26-30), ACM, New York, NY, 2003, 198-207.

[17] Microsoft. NetMeeting Home Page, http://www.microsoft.com/windows/netmeeting

[18] Mørch, A. Aspect-Oriented Software Components. In *Adaptive Evolutionary Information Systems*, N. Patel (ed). Idea Group, Hershey, USA, 2002, 105-123.

[19] Mørch, A. Three Levels of End-User Tailoring: Customization, Integration, and Extension. In *Computers and Design in Context,* M. Kyng & L. Mathiassen (eds.). MIT Press, Cambridge, USA, 1997, 51-76.

[20] ObjectWeb, JAC Project. http://jac.objectweb.org/index.html

[21] Popovici, A., Alonso, G., Gross, T. Just-in-Time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of AOSD 03* (Boston, USA, March 17-21). ACM, New York, NY, 2003, 100-109.

[22] Rohall, S.L., Patterson, J. Another Look at Replicated-Application Sharing, . Position paper for *CSCW 04 workshop on Making Application Sharing Easy: Architectural Issues for Collaboration Transparency* (Chicago, USA, Nov. 6-10). ACM, New York, NY, 2004.

[23] Veit, M., Herrmann, S. Model-View-Controller and Object Teams: A Perfect Match of Paradigms. In *Proceedings of AOSD 03* (Boston, USA, March 17-21). ACM, New York, NY, 2003, 140-149.

[24] Wilson,R. IBM Workplace Client Technology Powering Managed Client Solutions, http://www.eclipsecon.org/EclipseCon_2004_TechnicalTrackPresentations/21_Wilson.pdf