Towards Synchronous Collaborative Software Engineering

Carl Cook Warwick Irwin Neville Churcher

Technial Report TR-03/04, June 2004
Software Engineering & Visualisation Group,
Department of Computer Science and Software Engineering,
University of Canterbury, Private Bag 4800,
Christchurch, New Zealand
{carl, wal, neville}@cosc.canterbury.ac.nz

Abstract

CAISE, a collaborative software engineering architecture, provides extensible real-time support for collaboration between participating tools and users. The architecture maintains a semantic project model constructed incrementally from software artifacts as they are developed; this model is used to determine the impact of changes at a semantic level. This information is relayed to developers, providing them with awareness of others' locations, and alerting them to potential conflicts and the need for closer collaboration. We use examples from CAISE-based tools to illustrate the potential of real-time collaborative software engineering to enhance awareness of other developers' actions.

1 Introduction

Software engineers work in teams, but typical development tools are effectively single-user. In this paper, we present a working framework that supports the real-time concurrent development of a shared set of artifacts within a software engineering project. Caise is an environment developed to explore the potential for combining features of Computer Supported Collaborative Work (Cscw), such as relaxed floor-control policies and awareness of others' presence and actions, with features of software engineering, such as complex persistent artifacts. At all times, Caise maintains a semantic understanding of the project's artifacts, allowing useful information to be leveraged during development such as code inter-dependencies and relationships between concurrently editing users.

In earlier work [6], we reported that we had expanded the architecture to accommodate the Java programming language, and that we intended to develop some industrial-strength collaborative tools. In this paper we focus primarily on the architecture and role of the Caise server. We introduce a number of client tools in order to illustrate some of the ways that Caise can enhance real-time collaboration. We also report new additions to the architecture, such as the adaptation of a versatile model of software, a project changes database and a restructured event model.

The remainder of the paper is structured as follows: In section 2 we advocate the discipline of Collaborative Software Engineering (Cse) research. In section 3 we put forward our argument that real-time collaborative tools are increasingly necessary to support today's software engineering tasks. Section 4 gives a detailed description of the Caise architecture, with a particular focus on the advances in the latest version. Section 5 provides several examples of new tools developed for the Caise architecture, including the integration of Borland's Together with Caise. Finally, section 6 outlines the future work, including additional proposed tool development, user and system evaluations, and the analysis and visualisation of user activity data.

2 Defining Cse

It is essential to collaborate within software engineering. We observe, however, that direct support for collaboration is absent from most software engineering tools such as editors, compilers and debuggers. Instead, support for collaboration is relegated to version control tools such as Cvs [3], Rcs [13] and SourceForge [2]. Version control tools are a highly successful mechanism for archiving multiple versions of software, but software engineering tools often use the same approach to convert the complex task of developing an application into a series of partitioned activities that may be individually worked on in isolation. This introduces numerous problems as inter-developer communication is restricted and programming conflicts are likely to occur during project rebuilds (section 3 provides an example of such a conflict). Even today, the current practice in software engineering is only to perform a regular global synchronisation of all code—typically known as the *nightly build*—in an attempt to prevent individual programmers' efforts deviating significantly from the project version.

To increase the degree of awareness within a software development team, several comparable frameworks and tools to Caise have been developed ranging

from project management tools to collaborative debuggers. For example, Tukan is a code editor for SmallTalk programs, and uses several metaphorical representations to alert programmers about potential conflicts with each other [12]. More recently, the Jazz toolkit has been introduced within the Eclipse IDE environment, providing all developers with version control information, as well as awareness of others' locations within Java programs [4]. Finally, Rosetta is another collaboration-aware tool, allowing users to design and document UML class diagrams in real-time over the Internet [8]. A comparative analysis of such work is provided elsewhere [5].

Tools such as those described elsewhere are often limited to a specific language, development task, or set of tools. We argue that for a CSE architecture to be of significant value to the community at large, it must not prevent programmers from using their favourite tools, nor restrict programming to a given language or development methodology. Only when CSE tools and frameworks become extensible do we expect the mainstream of developers to use them.

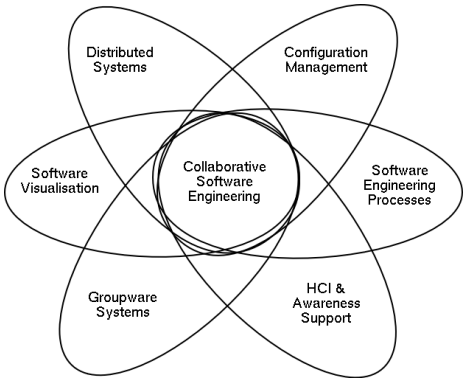

Figure 1: Research areas related to CSE

To provide genuinely extensible CSE tools, several distinct fields of research must be taken into consideration, including CSCW for groupware support, Human Computer Interaction for providing suitable forms of information display, Distributed Systems to assist interprocess communication, and Configuration Management to accommodate source code versioning issues. Whilst these fields are distinct in their own right, figure 1 illustrates the overlap between them and the field of CSE.

We observe that most tools developed to support CSE only exploit a few of the intersections between the related fields. In developing CAISE we seek to take into account all of these areas of research to provide an architecture that is truly extensible and general.

Unfortunately, we also observe that there appears to be no 'umbrella' field for CSE research; the relevant literature is fragmented. We argue that CSE deserves recognition as a distinct area of research that is critical to the advance of software engineering, particularly as complexity and size of projects increases.

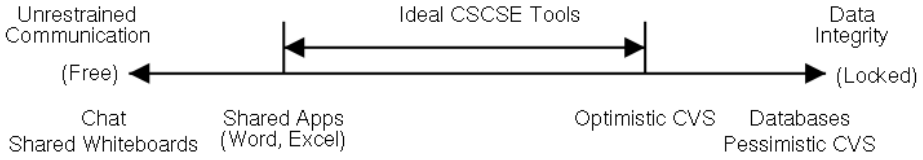

Figure 2: The collaborative spectrum in general software engineering terms

2.1 The Collaborative Spectrum

Software engineering can vary greatly in terms of degree of collaboration. In conventional approaches, such as version control systems, each artifact under modification is locked by its given developer. In order to reduce the granularity of code updates, optimistic locking is a possibility. This does, however, increase the complexity of merging changes when the underlying repository fails to consolidate the committed files—a situation that occurs frequently in real-world projects.

At the other extreme, developers could consider sharing every artifact via a CSCW tool. The goal of CSCW is to support fine-grained real-time interactions between participants by sharing the same application amongst several users, and by propagating each user's actions amongst all others by way of telepointers, telecursors and an array of other multi-user widgets. The GroupKit toolkit, for example, is excellent for developing collaboration-aware applications via CSCW technology [11].

Software engineering certainly involves times of fine-grained communication. Often, programmers work together on a single unit of code, and can benefit from the services of CSCW for chat and collaborative file editing facilities. Similarly, developers working on related units of code are also likely to initiate discussions, and will often be interested in what the other related developers intentions are. Unfortunately, CSCW technology does not provide a direct solution to supporting fine-grained collaboration within software engineering. Current CSCW technology is only applicable to relatively trivial applications when communication is mainly transient [9]; collaborative editing of highly structured, inter-related and persistent artifacts such as program source files is far beyond the scope of any CSCW system. CSE tools also have the difficult requirement that programmers must be able to revert back to the *last known good* version of a project, otherwise it becomes increasingly difficult to compile the system at regular intervals proportional to the number of concurrent developers—another requirement beyond conventional CSCW capabilities.

It is clear that in order for CSE to progress, we must find a balance between the potential for the fine-grained, casual interaction of CSCW, and the stringent requirements of software engineering such as the integrity and persistence of syntactically-structured source files. We feel that ideally, a CSE architecture should fall somewhere sensible between these two extremes, as indicated in figure 2. We do not want to jeopardize the integrity of the software engineering artifacts and products, but we also do not want the very low bandwidth of communication that is apparent with fully-locked source code control schemes.

3 Motivation for CAISE

Often programmers work concurrently, yet in isolation, using file locking and merging tools such as CVS to partition their set of active artifacts. Whilst this provides a mechanism for each programmer to work privately on a given task, such programmers often experience surprises when merging or checking out code, such as the code no longer compiling due to a simultaneous change to part of the project elsewhere by another user. In developing CAISE, we demonstrate that it is possible for tools to operate in a highly collaborative programmer-centric manner, where a single set of source files are browsed and edited collaboratively and in real time by any number of users and tools, even in a distributed setting.

The primary motivation in the development of our architecture is to increase the level of communication between project members and the timeliness of that communication. We argue that conventional software engineering tools, such as text editors, compilers and source code control systems unnecessarily stifle communication due to their single-user and artifact-centric approach. The development model imposed by such systems is one where appropriate levels of communication occur predominantly as a reaction to project compilation failure during the rebuild phase—problems that could have been avoided with earlier detection as a result of increased collaboration and communication.

Our goal is not to replace version control systems with architectures such as CAISE. We simply suggest that the use of version control systems should be restricted to what they were designed for—maintaining multiple versions of projects. We observe that the use of version control systems for managing fine-grained collaboration between highly-related artifacts is problematic, and that more semantically-aware mechanisms should be employed. We argue that architectures such as CAISE should be used to facilitate the fine-grained development of software, and version control programs used for coarser-grained duties.

3.1 Factors Influencing Collaboration

Conflicts arising from project modification may eventually lead to discussions and increased interaction between programmers to resolve the problem. Putting aside project-specific aspects such as the programming language used, the experience of programmers and team dynamics, conflicts and subsequent inter-programmer discussions are largely determined by the following factors:

Granularity This is both in terms of the size and number of artifacts locked by each user, and the granularity of the actual modification—a line, a method, or an entire class. The coarser the granularity of modification, the higher the likelihood of coding conflict.

Interleaving The more a series of modifications is interleaved by two or more programmers, the greater the chance of conflicts arising.

Semantic Understanding The likelihood of conflicts also increases when modifications are made to units of code that are known to be semantically inter-related. When developers understand that they are working between highly-coupled modules, greater care is taken to avoid breaking the code. Similarly, caution is often taken when developers are aware that a given unit of code is referred to by many others.

Awareness Support Locked files, CVS logs, and email are a common means of communicating the actions, intentions and status of others within a software engineering project, particularly in distributed development settings. At the other extreme, some editing tools today also provide a degree of CSCW support to increase awareness of others' current actions and locations in order to reduce the likelihood of conflicts.

3.2 A Development Scenario without CAISE

To illustrate the factors influencing inter-developer cohesion as identified in section 3.1, and to assert the necessity for architectures such as CAISE, we present a simple example of two developers working on a small Java-based project using only standard tools. The project's artifacts are held in a source code control system, and stand-alone code editors are used. The first developer, Wal, is working within the file 'DynamicSprite.java', as illustrated in figure 3. The second developer, Carl, is editing the file 'AnimiatedSprite.java'—which is a subclass of DynamicSprite. Both users employ the coarse-grained approach of taking copies of the entire source files needed to work on in private.

In the given example, there are no lexical conflicts—each class is in a different file, hence the concurrent editing of a single artifact does not need to be supported. Both users need to know, however, that the code they are editing is highly inter-related at the semantic level; in other words, they require a semantic understanding of the code base. In particular, Wal must be aware that certain changes to the implemented interfaces, constructors and properties could cause compiler errors or changes in implementation behaviour. Additionally, both users must be aware that changes to method signatures could also cause difficulties for Carl—specifically when explicitly invoked superclass methods and implicitly invoked superclass constructors no longer match.

At the very least, the actual area of significance for both users in the current scenario is both classes plus the parent class, as illustrated by the grey shaded area in figure 3. As other users begin editing classes that are semantically related to the three illustrated classes, the area of logical interest, or semantic understanding, grows—an area far beyond the set of observed artifacts for any one user. There are most certainly numerous other dependencies as well from other related classes, but in this example we are focusing on the direct dependencies between the artifacts being modified by Wal and Carl.

From the current example, we can see that the lexical units under the modification of Wal and Carl are highly inter-related. A modification to either class has the potential to significantly distort the other class, either in terms of a compilation error, or altered run-time behavior (for example, a changed signature to an overridden method of the subclass will change the actual method called). Unfortunately, the list of complications is not yet complete—we are yet to address development errors from Wal and Carl working on two different sets of source code.

To give an example of a simple, yet typical development error, let us imagine that Carl makes one publicly visible yet fine-grained change—he changed a parameter within a method from one type to another. He then updates another block of code that calls this newly changed method, and finally recompiles this code against the entire project. Everything works as intended—for now. At the same time, Wal begins work on his file by making a new call to the subclass's

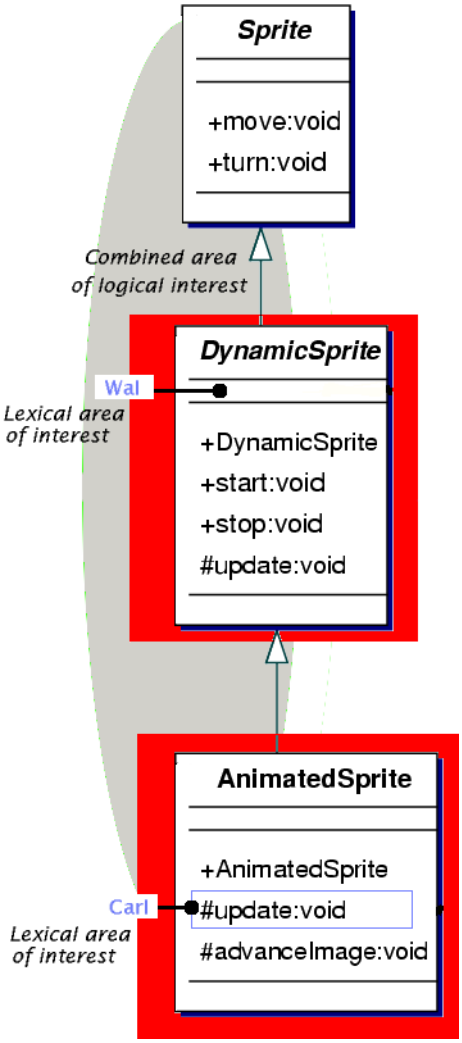

Figure 3: Lexical vs. logical views of code

method that Carl just changed—again, another fine-grained change. We know that Wal is now passing the wrong parameter type to this method, but the change by Carl is yet to be propagated to Wal via the source code control system. Accordingly, Wal's version of the project compiles as well. No one knows of this problem caused by interleaved modifications until the code is merged—at which point both programmers scratch their heads, and then begin their discussion on what went wrong and when.

The Caise architecture has been developed to prevent development problems by allowing programmers to discuss potential conflicts as they begin to form. In the above example, the Caise architecture would inform both users that they are in units of code semantically related to each other. Both users can then mediate through the communication channels provided by Caise, or perhaps enter another form of discussion to assess and resolve the current problem.

To provide useful information to each programmer, CAISE can identify what the type of code dependencies are involved with the programming conflict, the cardinality of the relationships, and if the change is accepted, what the impact on the project will be. At a CSCW level, CAISE will also provide both users' tools with awareness information related to the physical location of each user, and any actions such as code modifications.

3.3 Required Feedback Types

Readers can empathize that real-world programming difficulties resulting from concurrent editing are often far worse than the example given in the previous section, and are all too common. The longer the synchronisation intervals and the coarser the granularity of the locked code, the more problematic updates to code can be. Even worse, as the number of concurrent editors increases, difficulties are even more likely to arise. Additionally, many such changes happen simultaneously within real-world projects.

The types of feedback to prevent these problems, such as awareness of other user's actions, and awareness of relationships between semantically related units of code, are not provided by conventional tools. Initially it may appear relatively straightforward to integrate such support into existing software engineering tools, but to do so *correctly* requires an adequate understanding of the entire project, along with technology to control the editing of artifacts, monitor the activity of users, control the submission of changes and provide the relevant feedback.

The CAISE architecture defines and provides three main forms of feedback to increase the awareness of other developers' actions within a software engineering project:

Code Dependencies The references that other parts of the project make to a given lexical unit of code, and references that the current unit of code makes to others. For example, when an expression within a unit of code makes a reference to a method, a dependency exists between the two. For the expression, an outwards dependency exists, and for the method, this is an inwards dependency.

User Relationships These are relationships between the developers within the project. They are derived from the dependencies of the code being edited. User relationships are particularly important, as these identify the areas of the project that may experience conflicting changes.

Change Impact The set of resultant error messages and resolved errors based on a given modification. For example, if a new method call is added to a body of code, and the referred-to method does not yet exist, a message will be raised informing the user. Similarly, when a previously non-existent method is declared, a message will be raised that all prior references to it have now been resolved.

It is hoped that by providing this feedback support, we can avoid the problems of code synchronisation and programmer isolation.

4 The Caise Approach

We now discuss the implementation of Caise and how it supports the desired properties of collaborative software engineering tools as outlined in the previous section. With Caise it is immediately possible to merge the capabilities of Cscw with software engineering to support the real-time development of shared artifacts.

Caise is not a set of tools, although some Caise-based client tools are reported here. As illustrated in figure 4, Caise is an underlying framework to support collaborative software engineering tools. Caise keeps track of the software project as it changes, and provides feedback to participating tools. The architecture also handles the persistence of all software engineering artifacts within the project, maintains a change database of all significant changes made, and provides an interface to query and/or update the model.

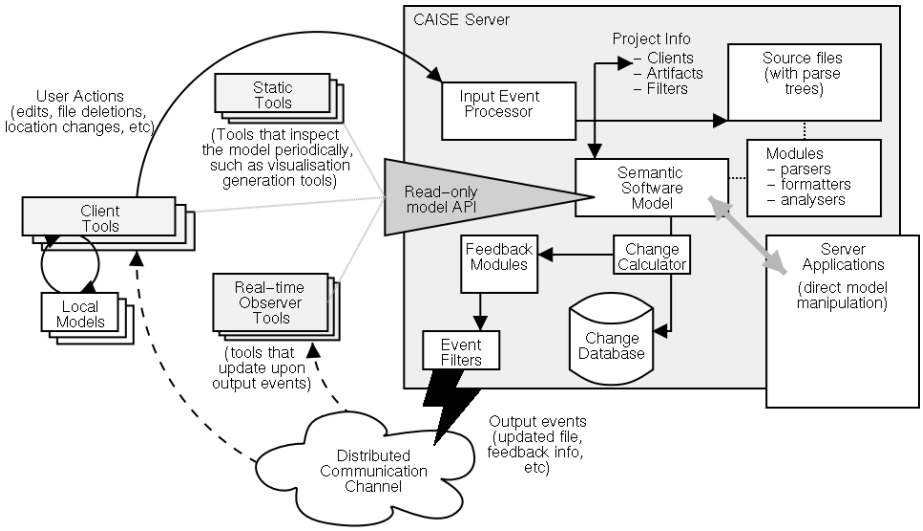

Figure 4: The entire Caise architecture, including participating tools

4.1 A Robust Model of Software

Caise now has a robust semantic model of software at the core of the architecture, as illustrated in figure 4. In the initial version of Caise, each language-specific semantic analyser maintained its own private model for each project implemented in that language. We realised, however, that Caise could be made more extensible if a common project model format could be used. After making this change, much of the work in maintaining the model is now provided by the server, making the development of language-specific analysers considerably easier. This also allowed us to generalise the event model, integrate a generic change determination module, and write language-independent feedback modules.

The semantic model is a fine-grained collection of all components within a software project, from packages and classes down to parameters, variables and

blocks. IDEs such as Borland's Together [7] and Eclipse [1] provide broadly similar models to ours in terms of program structure, however we also model the location of users within the project space, and incorporate a change history within each model component. The model is constructed incrementally via updated parse trees from recently edited artifacts, and can accommodate inconsistencies such as reference to undeclared classes and methods. Please refer to Irwin and Churcher [10] for a full description of the semantic model of software.

4.1.1 Related Work

The semantic model within CAISE is broadly similar to that of Eclipse. A significant difference, however, is that the model within CAISE is *collaboration-centric* where the one instance of the model per project is shared by all participating tools. In addition, the CAISE architecture and semantic model were designed upon the assumption that multiple programmers are likely to be editing a common set of artifacts at the same time. For a team of developers using Eclipse, each IDE instance has its own working version of an underlying project model. Therefore, each IDE instance must be manually synchronised at regular intervals by a version control system in order to maintain a degree of consistency within the project.

Additionally, as the semantic model maintained by CAISE is located within a server, it is affordable to hold the entire model in memory for unrestricted querying and manipulation by client tools and other components. Conversely, Eclipse is designed to run on standard workstations, therefore it must restrict the size of the in-memory software model in order to not use all the memory and processor resources. Therefore, direct access to the software model in Eclipse is not possible—access is instead provided by symbol-table indexes and a cached model.

4.2 The Server Architecture

The server within CAISE is responsible for coordinating events from all participating tools, maintaining the project's artifacts, and propagating events back out to all relevant listeners. From a client tool's aspect, the server is simply the point of contact for obtaining artifacts and other project-based information. The underlying duties of the server, however, are more complex than this, and can be summarised as *discovering dependencies* (building up the model), *identifying relationships* (inspecting the model) and *alerting change* (propagating events to relevant participants).

To describe the lifecycle of collaboration within the CAISE architecture, we refer back to the example scenario presented in section 3.2. When user Carl changed the parameter of a method from one type to another, a sequence of character-edit events will have been delivered to the server by Carl's CAISE-based code editor. Not only will Wal have been informed that Carl was located in a semantically related area of code, but Wal will now also see the file update in real-time if he chooses to view it in his code editor. Upon Carl's changes being recognised as syntactically correct, the CAISE server will re-parse the artifact, and update the model accordingly. The updated parse tree and source file buffer

will be propagated out to all artifact viewers, in which their local models can be updated if necessary.

In the case that Wal and Carl agree on the modification this time, the change to the model will be saved in the change database, and change information will also be sent to any other remote tools that are observing the project state. In the case that a third programmer wishes to open the same artifact at any time they are able to do so—the local artifact cache within each Caise-based tool is automatically updated with the real-time copies upon tool initialisation. For a more in-depth discussion of the Caise architecture including clients' local models, parse trees and event propagation, please refer to our previous paper [6].

4.2.1 The Change Database

A new feature in this version of Caise is the change database. This component of the architecture allows Caise to keep a record of all changes to the model—changes at the semantic level such as a renamed variable or a deleted class. The change calculator, working on the updated semantic model, performs a 'before and after' comparison to determine what has been added, removed and modified. This not only allows change events to be propagated out to all interested listeners, but gives researchers a complete breakdown of how a given project develops over time.

4.2.2 The Plugins Facility

The plugins facility within Caise is one of the key means of providing extensibility. Plugins are user-defined modules that extend the capabilities of the server.

By default, the Caise architecture provides basic support for sharing and storing both artifacts and the semantic model, and facilitates communication between all participating Caise-based tools. Language-specific plugins of the following types extend the architecture: *Parsers* allow artifacts to be translated from source code to a tree-based format in order to provide data for the semantic model. *Analysers* allow the model to be incrementally constructed and queried. *Formatters* provide a means of generating source files from a project model. *Feedback modules*, however, are general in nature because they operate purely on the semantic model, rather than with language-specific artifacts and parse trees.

4.3 Feedback for Caise-Based Tools

It is important to provide accurate information for developers, but this is difficult for complicated languages such as Java. Often, the only reliable information is derived from compiler and linker error messages. Caise provides accurate and reliable information because of its use of an underlying semantic model, which is comparable to that used by a compiler. Within Caise, feedback can simply be edit updates, user location updates, significant change updates, or customized feedback information based on user events.

4.3.1 Extending the Feedback

An important aspect of Caise is that feedback can be extended. Application developers can write module to perform custom calculations and feedback generation. Such feedback is sent to all requesting applications, providing the tools with the necessary data to fuel any desired type of feedback mechanism. Developers write a feedback module (or extend an existing one), place it in the server's plugin directory, and configure tools to catch and handle the resultant feedback events.

The type of action that Caise-based tools choose when feedback events occur is entirely at the discretion of the tool developer; Caise is simply the mechanism that provides collaborative management and feedback. New feedback modules can be written and used with Caise, providing virtually any type of information to Caise-based tools. Feedback modules are invoked upon relevant user events, at which point the project model may be inspected, including information related to the location of users within the project, the change history of the project and the entire project structure—from packages down to individual blocks of code.

4.3.2 The User Neighbourhood

The *user neighbourhood* is a simple mechanism for Caise-based tools to restrict feedback received from the server. Using the user neighbourhood component, programmers can instruct their tools to ignore atomic operations, such as character-by-character edits, made by other users. This gives programmers an opportunity to rebuild the complete project without risking compiler syntax errors due to the ongoing edits by others. Significant changes such as a new declaration being made, however, can not be ignored and will always be propagated; otherwise this will cause development errors to come into effect, as experienced in conventional version control systems.

Our approach throughout the development of Caise is always to deal with coding conflicts as they happen rather than ignoring them. We believe that the user neighbourhood mechanism keeps feedback within the acceptable range for fine-grained Cse.

4.3.3 The Event Model

The Caise event model now provides three core event types: user events, change events and feedback events. User events are those generated by tools when users edit code, commit a change, open, close or delete a file, or join or leave the project. Change events are notifications the server issues when it detects a change to the semantic model, as a result of an artifact being modified. Feedback events have been discussed previously.

In terms of utilising Caise events, client tools simply listen and respond to events they are specifically interested in. A code editor will be interested when a file is modified, whereas a project management tool might only be interested when an excessively high number of programmers are modifying one artifact. Finally, feedback events will only be propagated to tools that specifically request them. In section 5 we show how tools utilise such events.

4.4 Scalability

At present, the Caise architecture has only been tested with groups of up to five concurrent developers. We see no foreseeable reason, however, why the architecture will not scale to larger projects with many developers.

From our observations, the load on the Caise server has been lightweight—even though the server is just a standard dual-processor Linux/Intel machine which runs numerous other unrelated services in parallel. Under mainstream use the number of concurrent developers and project sizes could be considerably larger, however, work on semantically related artifacts typically involves only a small group of users regardless of the project size. Additionally, the critical work per project modification is not overwhelming; whilst the semantic model for an entire project may be vast, typically the area to inspect and process per modification is relatively small.

5 Applications

Fundamental to the motivation of Caise is that Cse tools and architectures should not be limited to a particular tool or programming environment. In this section, we briefly present two different software engineering tools, and describe how other tools can be integrated with the Caise architecture. In a later paper, we will discuss the integration and use of new Caise-based tools in more detail.

5.1 A Plugin for Together

A Caise-enabled version of Borland's Together Ide for Java is presented in figure 5. As Together provides a plugins interface, we implemented a Caise-based plugin that allows Together to integrate with the Caise architecture. Developers may now use this version of Together as their Ide when working on Caise-based projects, allowing all project artifacts to be shared and edited concurrently.

In the current example, user 'clc38' is editing the class 'DynamicSprite'. At the same time, users Wal and Neville are editing the subclasses 'MobileSprite' and 'Animated Sprite' respectively. To the users of Together, the most obvious impact of Caise is that updates to artifacts from remote users are immediately propagated to the local views of the Together Ide. There are, however, further enhancements in this version of Together to illustrate the way propagated Caise events support user awareness, as labelled on figure 5.

Under label 'A' is a collaborative project management widget that we have called a *change graph*. This panel provides real-time updates on the number of additions and deletions to the project's semantic model per active developer. The change graph listens to change events propagated by the Caise server as they occur, and updates its display accordingly. Whilst this component merely counts and displays changes of all granularities, countless complex project management components are possible and straight-forward to develop. Within the Together Ide, we decided to host the change graph in its own window, allowing users to hide it when desired.

Under label 'B' is another component to provide real-time feedback. Called the *user tree*, this panel keeps a running update on the locations of each active user within the project, and displays a 'viewing' or 'editing' icon according to

their status. Again, this component obtains its data from propagated CAISE events in order to show the location of users within the semantic model.

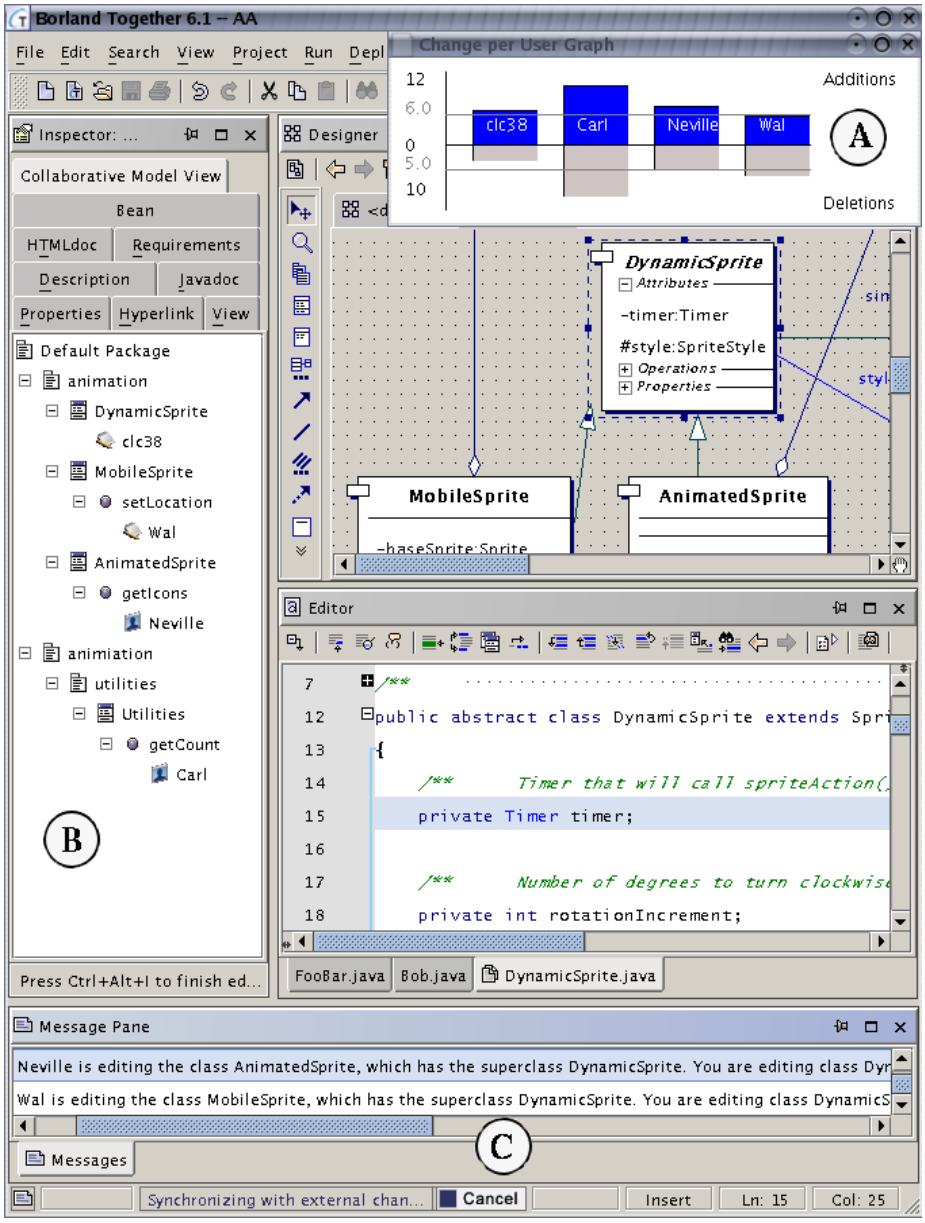

Figure 5: A CAISE-enabled Together IDE

Under label 'C' is Together's message panel. Again via Together's plugin facility, we modified the message panel to display messages from a specifically-written CAISE feedback module. This particular feedback module analyses user relationships between superclasses and subclasses, and warns pairs of users if such relationships between them exist.

The CAISE-enabled plugin for Together presents a comprehensive test of the

strength and viability of the Caise architecture. By using an industrial-strength tool such as Together, we can illustrate that Caise is suitable for commercial tool use, and can accommodate multiple users joining and leaving at any time.

5.2 A Code-Age Editor

Code age editors assist the visualisation of code 'lifetimes', where lifetimes typically mean the time since creation date for a given line of code, or the number of edits a line of code has received. As another example of how Caise supports collaboration by means of project change analysis and propagating events, in figure 6 we present a Caise-based code age editor for the Java language. To the best of our knowledge, this is the first collaborative, real-time code age editor for any programming language.

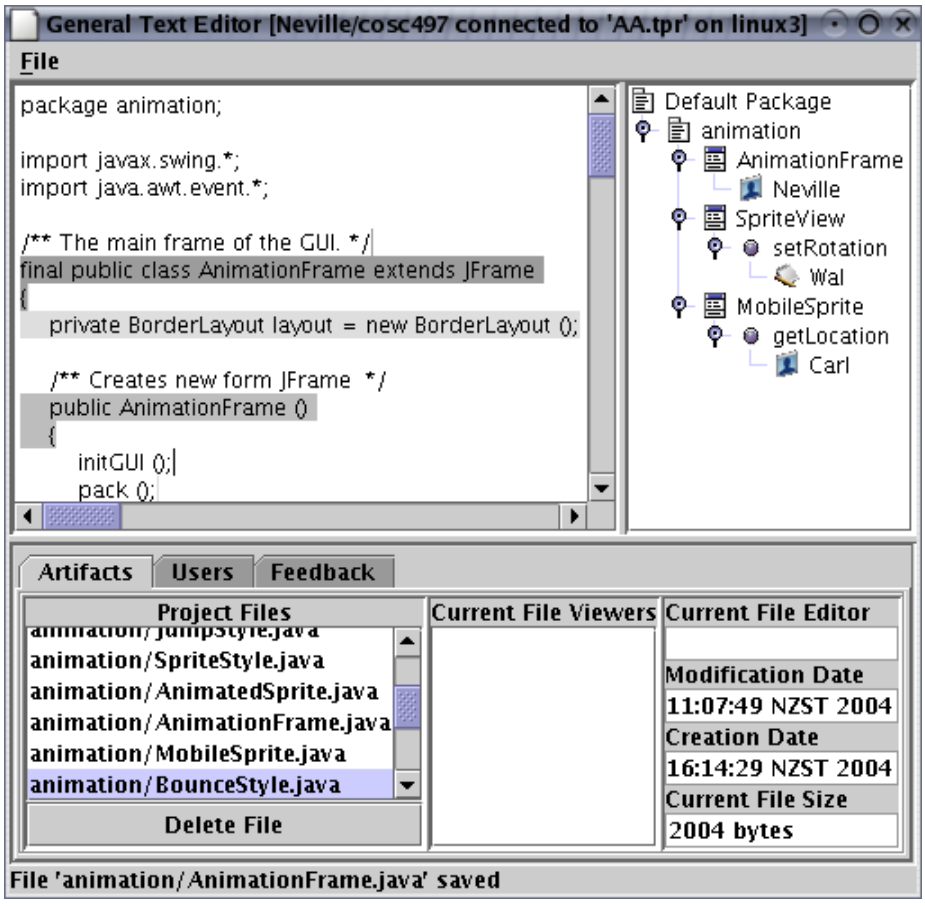

Figure 6: A code age editor for Java

The code age editor presented here has another interesting feature: code age is not on a line by line basis—it is *per component* because of the server's ability to detect fine-grained changes within the model. The code age editor updates its display every time a new project change event is received from the Caise

server; at which point the editor inspects the updated parse tree and extracts the modification history details for each component. Therefore, this code age editor can display the code age for each non-terminal production in the artifact—from a class declaration right down to a string literal. Upon identifying a component with previous changes, the editor shades the appropriate declarations according to their number of modifications.

The client panel, as described in our previous paper on Caise, is also included within the code age editor to provide additional awareness support. The client panel provides information on the project's artifacts, facilitates communication amongst project members via both voice and text chat, and provides an area to display custom feedback messages as they occur.

The final component within the code age editor is the user tree. The user tree gives an object-oriented model-based view of the project, focused on the locations of the current users. This type of awareness is useful even within comprehensive tools such as Together, but for an otherwise naive application such as the code age editor, semantically rich information is now readily available. Even though the editor's knowledge only extends to displaying and shading text files, the user tree provides full information about the underlying model—how it is constructed and where everybody is within it.

5.3 Integration of New Tools and Languages

To support a new language within Caise, three language-specific plugins must be supplied: a parser that generates Caise-compliant parse trees, an analyser to assist in the building and querying of the semantic model, and a formatter to generate source files. The parser and formatter can normally be produced by generator programs, and the analyser, whilst relatively complex, provides an intuitive Api to assist development.

Any software engineering tool can be extended to utilise Caise once the relative language(s) have been integrated, allowing it to be collaboration-aware. Once a tool has been written or converted—simply by making appropriate calls to the Caise Api—it can share and edit project artifacts, and respond to Caise generated events. Any number and type of Caise-based tools can work together collaboratively with a project. For example, the code age editor and Together Ide were used concurrently on the same project during our evaluations and testing.

Individual Caise-based components such as collaborative widgets can also be developed by utilising the Caise Api. For example, the change graph, user tree and client panel are all extensions of Swing JComponents, and can be added as a widget to any existing Java application to make it collaboration-aware, without the need for any further modification or configuration.

6 Conclusions and Future Work

The Caise architecture provides a basis for supporting Cse in which developers located anywhere on the Internet can work together effectively. Our approach combines the immediacy and flexibility of Cscw with the rigour and robustness of conventional development tools, and draws from other fields, such as information visualisation, as needed.

In this paper, we have described the design and implementation of the Caise server, which manages the interleaving of concurrent developers' activities via incrementally-updated parse-trees and a corresponding semantic model.

The server also generates events, based on its analysis of the semantic model and the actions of users, which are then propagated to individual tools. In this way, individual tools (and hence users) are made aware not only of the locations and actions of others, but also of potential conflicts.

Such awareness allows users to work more closely (supported by chat, voice or video communication) where potential conflicts arise, yet to assist productivity by permitting increased fine-grained interleaving where appropriate.

We have also illustrated the application of Caise by discussing some specific client tools which utilise feedback in various ways. The Caise change database is a valuable resource for other tools, particularly those performing off-line activities, such as design critics or process metrics analysis.

We are encouraged by the successful implementation of Caise and client tools. Performance is adequate where both server and client tools are running on standard machines. Feedback from focus group trials is also encouraging.

Our work on Caise is part of a long-term project and many aspects of Cse are yet to be explored. As our tools mature, we are in a position to conduct both longer-term experiments, to compare the effectiveness of collaborative development with traditional approaches, and shorter term Hci and other evaluations of specific features such as the Gui components which deliver user feedback.

Our future work includes aspects such as the following:

Metrics and Analysis The Caise logs provide data concerning the ways in which developers collaborate and the frequency of changes to individual artifacts. Analysis and visualisation of this data can shed light on development practice and product quality issues.

Feedback Too much feedback is overwhelming; too little is dangerous. Useful feedback is context-sensitive; sometimes it should be unobtrusive, yet sometimes needs to provide an alarm.

Patterns of Collaboration Sometimes a group will 'follow the leader' around a project, but may then switch to working in relative isolation, followed by concentration of changes in a small number of artifacts. In processes such as eXtreme Programming, close collaboration between developers may be critical.

Client Tools We aim to identify those tools which can benefit most from Caise and to develop a suite of tools which includes stand-alone tools as well as plug-ins for Ides such as Eclipse.

Performance and Scalability There may be natural limits on the size and interaction of groups of developers; there will always be technical limits to what can be supported. Longer-term studies may indicate optimal group and project sizes.

References

[1] Eclipse Platform Technical Overview Version 2.1. White Paper, February 2003. URL `www.eclipse.org/articles/`.

[2] SourceForge.net Home Page. Internet URL, July 2003. URL `www.sourceforge.net`.

[3] B Berliner. CVS II: Parallelizing Software Development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[4] Li-Te Cheng, Susanne Hupfer, Steven Ross, and John Patterson. Jazz: A Collaborative Application Development Environment. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 102–103, Anaheim, California, USA, October 2003. ACM Press.

[5] Carl Cook. Collaborative Software Engineering: An Annotated Bibliography. Technical Report TR-COSC 02/04, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, June 2004. Work in Progress.

[6] Carl Cook and Neville Churcher. An Extensible Framework for Collaborative Software Engineering. In Deeber Azada, editor, *Proceedings of the Tenth Asia-Pacific Software Engineering Conference*, pages 290–299, Chiang Mai, Thailand, December 2003.

[7] Christopher Garrett. Software Modeling Introduction: What Do You Need from a Modeling Tool? White Paper, 28 May 2003.

[8] Nicholas Graham, Hugh Stewart, Authur Ryman, Reza Kopaee, and Rittu Rasouli. A World-Wide-Web Architecture for Collaborative Software Design. In *Software Technology and Engineering Practice*, pages 22–32, Pittsburgh, Pennsylvania, August 1999. IEEE.

[9] Jonathan Grudin. Groupware and social dynamics: Eight challenges for developers. In *Communications of the ACM*, volume 37 of *1*, pages 92–105. ACM Press, January 1994.

[10] Warwick Irwin and Neville Churcher. Object Oriented Metrics: Precision Tools and Configurable Visualisations. In *9th International Software Metrics Symposium*, Sydney, Australia, September 2003.

[11] Mark Roseman and Saul Greenberg. Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, March 1996.

[12] Till Schummer. Lost and Found in Software Space. In *34th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January 2001. IEEE.

[13] Walter F. Tichy. RCS — A System for Version Control. *Software — Practice and Experience*, 15(7):637–654, 1985.