

Constructing Real-Time Collaborative Software Engineering Tools Using CAISE, an Architecture for Supporting Tool Development

Carl Cook

Neville Churcher

Software Engineering & Visualisation Group, Department of Computer Science & Software Engineering,
University of Canterbury, Private Bag 4800, Christchurch, New Zealand
E-mail: {carl,neville}@cosc.canterbury.ac.nz

Abstract

Real-time Collaborative Software Engineering (CSE) tools have many perceived benefits including increased programmer communication and faster resolution of development conflicts. Demand and support for such tools is rapidly increasing, but the cost of developing such tools is prohibitively expensive. We have developed an architecture, CAISE, to support the rapid development of CSE tools. It is envisaged that the architecture will facilitate the creation of a range of tools, allowing the perceived benefits of collaboration to be fully realised. In this paper, we focus on the development of CSE tools within the CAISE architecture. We present examples to illustrate how such tools are constructed and how they support real-time multi-user collaborative software development. We also address issues related to the number of collaborators and discuss performance aspects.

Keywords: Collaborative Software Engineering, CSCW & Groupware, Tool Construction, Continuous Integration

1 Introduction

Software engineering is a predominantly collaborative activity. Typically multiple teams of people develop and maintain successive versions of a range of products, in parallel. Surprisingly, tools to support synchronous, or real-time Collaborative Software Engineering (CSE) are still restricted to minor tasks for specific software engineering purposes—if they make it out of the research prototype stage at all.

Today there is a real need for CSE tools, and this demand will grow as software engineering becomes an increasingly complex and heterogeneous discipline. While support for collaboration has emerged in other areas of everyday applications such as file sharing, instant messaging, and generic tele-working, software engineers themselves appear ambivalent about the opportunities and potential benefits of more comprehensive tool support. Accordingly, research into CSE is both timely and imperative.

The main premise of our research is that by enabling fine-grained CSE through seamlessly integrated tool support, it is possible to raise the very restricted levels of communication within current software engineering practice. The value of active communication has long been recognised in Computer Supported Collaborative Work (CSCW) re-

search (Greenberg 1989), and is re-emerging in software engineering within the eXtreme Programming movement. Increased programmer communication, as put forward current CSE research, is likely to produce more informed decisions during the development stage of software engineering, and less likelihood of costly coding conflicts.

In this paper, we discuss how new tools can be constructed within the CAISE architecture to support the real-time development of a collaborative software project. We also address issues related to large group sizes and performance aspects of the architecture. The design of the CAISE architecture has been described previously (Cook, Churcher & Irwin 2004).

The remainder of the paper is structured as follows. Section 2 provides a background related to real-time CSE and supporting tools. Section 3 provides an overview of the CAISE architecture with particular emphasis on the services it provides to the construction of new CSE tools. Section 4 presents some typical CAISE-based CSE tools in use today, and Section 5 illustrates how to build new CSE tools within the CAISE architecture. Section 6 addresses the performance of the CAISE architecture and supporting tools, and a summary is given in Section 7.

2 Background and Motivation

In the last year many of the major commercial IDEs have taken significant steps towards code-level real-time collaboration. Of the five Java IDEs that have the largest market shares, Eclipse, Borland's JBuilder and Sun's JSE now support shared development facilities, and all vendors are promising more to come in the next major releases.

While the proposal of tools to support CSE often draws an enthusiastic response from practitioners, the design and implementation of commercial-strength tools is a challenging task. Even once such tools have been developed, there is no guarantee that they will gain widespread adoption; this mistake has been made within related areas of CSCW research (Carasik & Grantham 1988).

Any CSE tool has complex issues to address, such as user interface design, CSCW floor control and management, varying levels of collaboration requirements, varying expectations between developers within a group, support of multiple artifact types, and potentially multiple views of artifacts. There are also technical aspects to address such as concurrency control and distributed system design, along with the standard software engineering technicalities such as parsing, semantic modelling and source code management.

Very few research prototypes have evolved into features within professional tools. A significant difficulty is that conventional software engineering tools are designed for single-developer use, and 'bolting on' col-

laborative features does not necessarily scale or provide the level of improvement envisaged.

Implementing collaborative features for commercial strength tools, as we and others have discovered, is a very challenging problem. To date, software engineering tools typically work with the lowest common denominator of software engineering artifacts: source code. By employing source files as the finest-grained type of shared information, and by supporting information sharing only through code repository systems, it is difficult to extend IDEs to support real-time within-files collaboration, to provide support for additional views of software, and to provide collaborative access to the underlying software model.

While it is certainly possible to implement collaboration-enhanced real-time software engineering tools, the single significant barrier to the success of tools may be the poor ratio of tool power to development effort.

2.1 Why CSCW Fails for CSE

An initial solution to implementing CSE tools is to use CSCW Groupware, and this has been trialed elsewhere with varying degrees of success (Schummer 2001) and (Churcher & Cerecke 1996). Groupware toolkits such as GroupKit (Roseman & Greenberg 1996) and Maui (Hill & Gutwin 2003) allow the sharing of text documents, whiteboards and other common forms of electronic media, and good results have been achieved when converting single user generic applications to their multi-user equivalents (Cox & Greenberg 2000).

Problems occur, however, when building industrial-strength CSE tools from Groupware toolkits. Professional tools are not limited to a single task, a single language or a single artifact view, and this is orthogonal to the characteristics of Groupware support. Additionally, Groupware has no understanding of the complex semantics or syntax of artifacts, and the relationships between users and artifacts. Software engineering artifacts are also persistent over a long time scale, whereas Groupware is more aligned to the support of transient communication with throw-away artifacts.

An attempt could be made to extend a single-user IDE collaboratively through the use of a CSCW toolkit, allowing it to support distributed collaborative development of code or UML diagrams. However, extending it where multiple views of artifacts are supported at the same time, such as round-trip engineering between source code and UML diagrams, is extremely difficult. This is particularly true if the main source of information is derived from the source code management system; shared access to an in-depth semantic model of the project is typically required to translate between different views. Commercial IDEs do not provide this functionality adequately, and IDE model sharing is only in its infancy within the Eclipse platform via the Eclipse Communication Framework project (Lewis 2005).

Given that CSCW technology does not scale to meet the needs of CSE, and IDEs do not provide enough fine-grained information to support the development of highly-synchronous new tools, often the only means of producing new tool sets is by completely redesigning tools upon a foundation of collaborative software engineering technology.

2.2 Related Work Towards Real-Time CSE

The augmentation of software engineering tools with Groupware features failed to produce tools of any considerable impact within the field of research.

A decade on since the conception of Groupware toolkits, researchers have turned to implementing collaboration-based prototype tools upon existing software engineering frameworks.

Palantír, for example, is a system that inspects in real-time the impact of changes within source code repositories (Sarua & van der Hoek 2002). Rosetta provides an Internet-based collaborative class diagramming tool-set (Graham, Stewart, Ryman, Kopae & Rasouli 1999). Tukan, using the COAST Groupware framework, is a shared code editor for SmallTalk (Schummer 2001). Augur provides comprehensive user activity visualisations based on source code management systems (Froehlich & Dourish 2004). Moomba is a recently published system for supporting distributed eXtreme Programming (XP) within a global software development context (Reeves & Zhu 2004).

Tools such as those listed above are well suited for a single task or development methodology, but they are for the most part fixed and non-extensible. Additionally, it is certain that each tool took a considerable amount of effort to design and implement. The purpose of our research is to reduce the barrier of high construction costs for CSE tools by proving a architecture that enables many types of CSE tools to be developed rapidly.

3 The CAISE Architecture

The CAISE architecture, as illustrated in Figure 1, allows isolated programmers to work collaboratively without sacrificing communication. CAISE-based tools achieve this by keeping all programmers within a group synchronised in real-time, at the same time providing customisable user awareness and project state information to the individual tools.

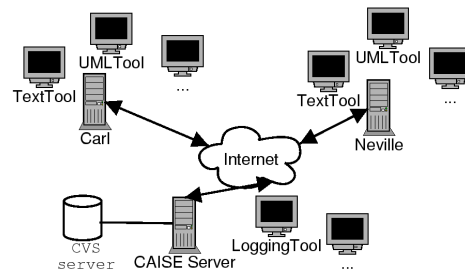


Figure 1: A schematic representation of CAISE.

The CAISE architecture provides an infrastructure with the potential to support the entire software engineering process. CAISE tools can be constructed that provide more than just the shared editing of basic software artifacts. Collaborative compilation, testing and debugging of software projects are also possible to implement using the services of CAISE. Comprehensive inter-developer communication facilities can also be constructed.

The CAISE architecture is not built on top of a source code repository system. CAISE and its supporting tools, however, do not aim to replace source code repositories either. The ability to work in private at times and to be able to keep different versions of programs separate are elements that very few programmers could do without. Our tools, therefore, are designed to support what code repositories do not provide: communication between developers and tools during fine-grained real-time collaboration, such as multi-user coding within the same file.

The example CSE tools presented in this paper support the core functionality expected of any software engineering suite, including project compilation

and execution support, editor undo, cut, copy and paste, UML class diagramming, and round-trip engineering between tools.

To date, the current set of CSE tools have performed well anecdotally (Cook et al. 2004), empirically (Cook & Churcher 2005a), and heuristically (Cook & Churcher 2005b). The underlying CAISE architecture allows for the rapid development of fully featured CSE tools, such as those presented in this paper. Typically, CAISE tools are designed to support patterns of collaboration evident in software engineering practice. Such patterns are presented in (Cook 2005).

3.1 Comparison to Other CSE Architectures

We are aware that different programmers will have different tool requirements, and we make no assumptions about the ‘right’ set of software engineering tools. Accordingly, CAISE has been implemented as an extensible architecture rather than a tool-set. We have focused on designing a architecture that can support a wide range of custom collaborative applications.

In contrast to most other tools, CAISE employs a holistic approach in that the entire infrastructure is based upon collaboration. At the core of the architecture lies a shared semantic model of the software being developed, allowing multiple views of artifacts to be supported by any number of different tools. The CAISE server, which houses each project’s semantic model, exists as a shared IDE engine for collaborative tools. This is depicted in Figure 1, where all management, parsing and semantic analysis of shared artifacts is provided by default.

The semantic model housed within the CAISE server is primarily focused on object oriented languages. Full support is available for Java 1.4, with work near completion for Java 1.5, including Generic Types. Other ‘Java-like’ languages such as C# can be supported by inserting a new semantic analyser into the CAISE server. Such an analyser maps C# parse trees into a model-based program structure, which the CAISE server integrates into the project’s semantic model.

CAISE is not simply a collaborative add-on project to existing single-user tools. CAISE operates as a fine-grained Continuous Integration (CI) platform where all tools are synchronised with each other, as opposed to systems that periodically attempt to resynchronise through source code repositories and social protocols. The CAISE architecture also differs by being capable of supporting complex CSE tools. It is not restricted to particular tasks or methodologies—there is no theoretical limit to the scale and ability of CAISE-based tools.

3.2 Tool Services

CAISE provides services which support the rapid development of CSE tools. By utilising the CAISE architecture, CSE tools can rely on the CAISE server to manage the storage and sharing of artifacts, and to control users as they join and leave projects and artifacts. CAISE also provides the low-level mechanisms to allow distributed messaging between tools and the CAISE server, and supports a distributed event model.

A semantic model of the software for each project is maintained by the server, which is refined upon the actions of participating CAISE tools. The semantic model represents the collection of components within the software project, such as packages, classes, methods, properties, and relationships such as inheritance,

association and invocation. CAISE-based tools are not required to perform any parsing or semantic analysis themselves; the server is responsible for translating modifications in artifacts to an updated semantic model. The model, however, is accessible by CAISE tools both for reading and direct modification if required.

The functions provided by the CAISE server, both in terms of supporting collaborative work and performing core software engineering tasks, allow the CSE tool developer to focus on the specific requirements of the given tool rather than re-implementing the functionality common to most CSE tools. If, however, the tool being developed requires additional features, the CAISE architecture is easily extended to accommodate new artifact types and kinds of feedback. Section 3.7 discusses this concept further.

3.3 CAISE Tool Widgets

Before we look at the construction of individual CAISE-based tools, this section introduces some of the collaborative widgets available for use within any CSE tool. The following widgets can be added to Swing/AWT-based Java applications without any additional coding requirements, which allows tools to be augmented with CSE capabilities for minimal effort. These widgets operate by communicating with the CAISE server and responding to real-time events.

The *User Tree* is shown in Figure 2, which may be used within a tool to support user awareness. This widget provides a user-centered view of the CAISE-based software engineering project in real-time. Individual tools require no knowledge of the artifacts they are editing; they simply have to keep the CAISE server informed of the name of the artifact currently being edited and the most recent cursor location of the user controlling the tool. The User Tree will keep itself updated with the latest view.

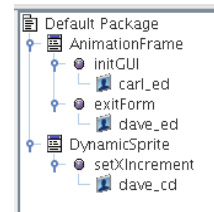


Figure 2: The CAISE *User Tree* widget, supporting a user-centric project view.

The *Change Graph* is another widget that can be readily added to any CSE tool. This widget is illustrated in Figure 3. The Change Graph widget keeps track of the cumulative additions and deletions to and from the model on a per-user basis. This provides each tool user with an overview of the current development activity. Again, this component can be added to any CSE application, or housed in a dashboard display or separate frame.

The *Client Panel* is key component of the CAISE widgets package, and can be seen within the CAISE-based tool presented in Figure 7. The Client Panel typically houses four components known as the *Artifacts*, *Users*, *Feedback* and *Build Panes*, although the Client Panel can be configured to house any combination of specific panes. Individual panes can also be added to an application separately.

The Artifacts Pane is presented at the bottom of Figure 7. This provides basic file information on the artifacts within a CAISE project, including their current compilation state. The Users Pane is presented in Figure 5. This pane allows messages to be sent between users, including audio broadcasts.

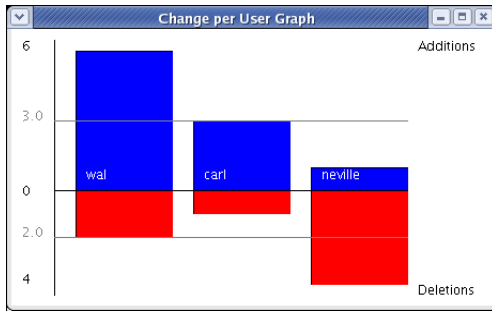


Figure 3: The project *Change Graph* widget.

The Build Pane is presented in Figure 4. It provides an adjustable level of collaborative awareness, allowing the user to temporarily ignore concurrent edits for the purpose of building the system without interruption. In addition to allowing the project to be built from the live, last parseable or last buildable version, the Build Pane also allows the current project to be executed. Finally, the Feedback Pane, which is not presented in this paper, displays server-based plain-text information such as Degree of Awareness (DOI) feedback between users and impact reports that result from artifact modifications.

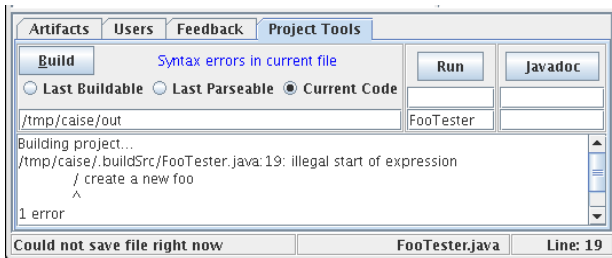


Figure 4: The CAISE *Build Pane* with adjustable levels of collaborative awareness.

The CAISE Tool widgets can be utilised from within any application, and applications that use such components do not require any specific software engineering knowledge or capabilities. For example, a stand-alone text editor can be enhanced by incorporating the CAISE collaborative User Tree into its user interface. Given that the text editor conforms to the CAISE Tool Protocol as specified in Section 3.5, the User Tree will highlight the method, class and package that the editor is currently modifying, without the editor needing to possess any specific software engineering capabilities.

Within the next release of CAISE, we also aim to add the multi-user text pane to our collaborative widgets package. The text pane is illustrated in Figure 7.

3.4 The CAISE Tool API

The *CAISE Tool API* (CTA) is provided as the means of accessing the functions of the CAISE server from within a CSE application. While the CAISE server typically resides on a separate machine, the CTA allows the calling application to view the server as if it was contained within the same process; the server functions appear no different to those of any other library. The server is accessed by a set of standard method calls, data is marshalled as method return values, and catchable events are thrown whenever interesting actions occur during the development of a CAISE project.

Table 1 presents several of the key CTA methods. This is only a subset of the complete CTA, but it provides a useful overview of the programming interface.

The CTA provides adequate functionality to implement numerous types of CSE tools. Multi-user text editors, for example, can rely on the CTA to provide collaborative code editing, semantic analysis of code modifications, and user presence feedback. To implement communication facilities, messaging can be provided via the Chat methods, and audio broadcasts are also supported. Tool design and implementation will always be the responsibility of the CSE researcher, but the CTA prevents ‘reinventing the wheel’ for the essential yet complex CSE services.

CAISE is a concurrent system, where it is likely to receive multiple interleaved requests to modify a set of artifacts within a short period of time. Invocations of CTA methods are treated fairly at the CAISE server. In the underlying distributed system that CAISE employs for its client interface, each incoming method call is queued and then processed in a thread safe, sequential order. For all other pending method calls in the queue, a low-CPU blocking mechanism is used on the client side.

3.5 The CAISE Tool Protocol

By following the *CAISE Tool Protocol*, which specifies the contract between individual tools and the server, tools are assured of staying synchronised with each other, and the server is able to avoid concurrency issues such as deadlocks and forced rollbacks of tool requests.

Individual CSE tools have the ability to implement locks and other floor control policies that allow only one user at a time to edit a given region of code. By default, however, the CAISE architecture allows full synchronous editing of any artifact. To ensure that tools are always synchronised, a specialised Model-View-Controller (MVC) approach is used which guarantees consistency over distributed parallel edits. Requests to edit the view are captured by tools, but the view is not immediately updated. Rather, the edit is sent to the server which in turn edits the global model, and broadcasts the change to all tools. Each tool then updates its local view, including the tool that made the edit request.

To implement a CSE tool that adheres to the CAISE Tool Protocol, three application-level threads are typically used: a GUI thread, a worker thread, and a CAISE event listener thread. The threading model for CAISE-based tools is presented in Figure 6. Most windowing toolkit libraries provide a GUI thread, and the CAISE Tool API provides a CAISE event listener thread. As the worker thread can simply be the main application thread, it is unlikely that any new threads need to be created explicitly within a CAISE tool. With the existence of a worker thread, the GUI thread is free to take any volume of user input from the user interface, without causing jitter or lag as the events are being proxied to the CAISE server.

By using a MVC approach and following the CAISE Tool Protocol, CSE tools are guaranteed to stay up-to-date and synchronised with the CAISE server, and there is no risk of deadlocks or loss of information. The following list presents the six key conditions of the CAISE Tool Protocol:

1. The CSE tool captures all user input events such as keystrokes and caret move events, typically using action listeners. All actions are to be consumed, blocking the underlying view of the artifact from being modified.
2. All captured events are placed into a FIFO event queue within the CSE tool. The GUI thread returns immediately after placing the event in the

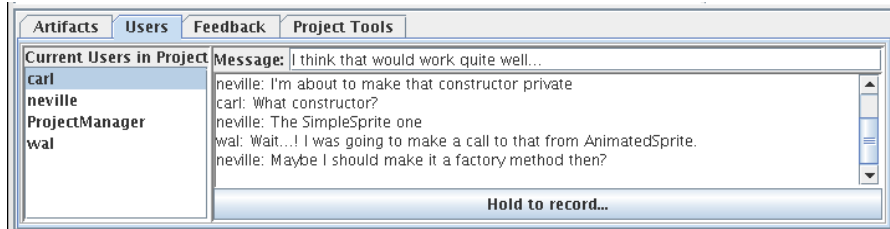


Figure 5: The CAISE *Users Pane*, which provides voice and text communication.

Method	Description
Connect to Engine	Makes a new connection to the given CAISE server
Open Project	Opens an existing CAISE project
Add Artifact	Adds a new artifact to the given project
Open Artifact	Sets an existing artifact as open for a given user
Set User Location	Moves a user's cursor location within an artifact
Update Source Code	Appends a sequence of characters to an artifact
Update Parse Tree	Appends a parse tree of an artifact
Update Model	Directly manipulates the semantic model of a project
Get Model Snapshot	Returns a copy of a project's semantic model
Fire Tool Event	Allows a tool to invoke tool-specific server plug-ins
Get Event Log	Returns the complete event log for a given project
Send Chat Message	Allows users to send text messages between tools

Table 1: Key methods of the CAISE Tool API.

queue, preventing any latency within the user interface.

3. A separate CSE tool worker thread dequeues events and issues them to the server as corresponding CTA method invocations.
4. The CSE tool worker thread waits for the return value of the CTA method invocation before processing the next tool input event. The CSE tool does nothing upon a successful method invocation, and escalates any errors if the method invocation fails.
5. The CSE tool's CAISE event listener thread listens for broadcasted server events that result from CTA method invocations. Upon relevant events such as artifact modifications and user location changes, the model of the artifact within the tool is updated accordingly. This step is performed by all participating tools, not just the instance that invoked the event.
6. Upon any model update, the CSE tool's artifact view is redrawn by the GUI thread.

During spikes of development by multiple CAISE tools, the server ensures fairness by queuing events evenly based on the inbound tool connection, rather than order of arrival. In this manner, we avoid the situation where all other tools are unfairly delayed by an exceptionally active single user.

3.6 Tool Events

Within the CAISE event model, the CAISE server broadcasts events of various types to all participating tools upon actions of individual users; this is illustrated in Figure 6. Events contain details of the general action, such as an artifact edit by a named user, and the specific details such as the affected text and file offset.

Each CAISE event type is briefly summarised in Table 2. Fully featured tools are likely to register as a listener for all events, as can be seen in the code listing in Section 5.1. Other components, such as the Change Graph presented in Section 3.3, are only interested in specific event types.

Type	Typical actions
Project	A project is created or deleted.
Artifact	An artifact is added, removed or edited.
Chat	A user issues text or audio messages.
Client	A client opens, closes or moves location within an artifact, or rebuilds a project.
Change	The project model is manipulated directly or via artifact modification.
Plug-in	Tool-specific custom units of information exchange.

Table 2: Events types within the CAISE architecture.

3.7 Tool Manager Modules

The CAISE architecture provides generic support for the collaborative editing of text documents, the parsing of source files, and the semantic analysis of parse trees derived from source code and UML diagrams. Often, however, tools require further functionality from the server, including the support of new artifact types. To accommodate extensibility within the CAISE server, modules known as *Tool Managers* can be integrated through the CAISE plug-ins interface.

For a UML class diagrammer, it is apparent that such a tool requires information beyond what is contained within the core semantic model of the project. As well as displaying all classes, methods and relationships, a class diagrammer contains class layout information that must be shared every time any instance of the class diagram is modified. Therefore, when implementing the UML diagramming tool presented in this paper, an additional diagrammer-specific type of artifact was introduced, managed and shared by a UML-specific Tool Manager.

For the UML diagramming tool, whenever a user changes the location of a displayed class, a tool-specific event is thrown to the CAISE server via the CTA and this event is proxied to the UML diagrammer Tool Manager. The Tool Manager will then access and update the new artifact that stores the class location information, and then broadcast this change out to all tools in accordance with the CAISE Tool Protocol, allowing all users to update their local view of the UML class diagram.

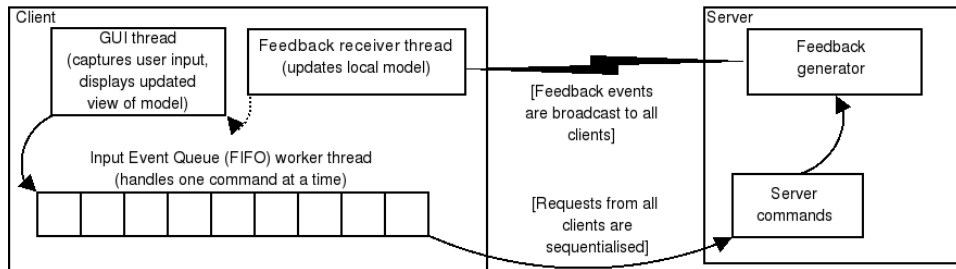


Figure 6: The recommended threading model within a CAISE-based tool.

The CAISE server can be extended in many other ways through the plug-ins interface, including support for new languages. For further details please refer to (Cook et al. 2004).

4 Example CAISE-Based Tools

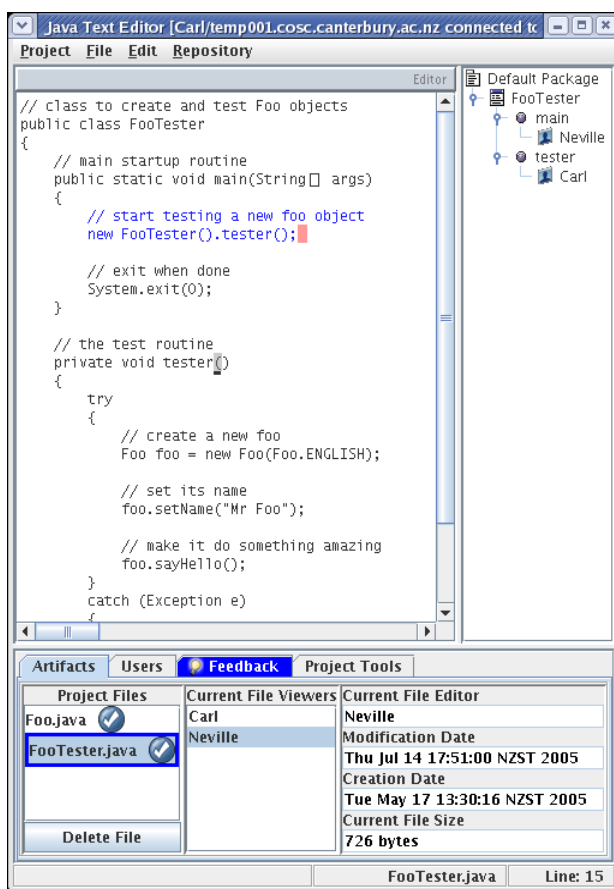


Figure 7: A CSE code editor. When viewed in colour the text highlighting and tele-cursors are visible.

Since their conception, we have been constantly refining the CSE tools discussed in this paper in order to provide realistic Software Engineering environments. These tools are presented in Figures 7 and 8. The main features currently supported include:

- Round-trip engineering between all tools and artifact views.
- Full multi-user editing and UML class diagramming capabilities with a *relaxed WYSIWIS* view, including collaborative undo.
- An Artifacts Pane that displays the current compilation state of each artifact as well as editor details and file information.

- A multi-user text pane which provides remote modification highlighting and *tele-cursors*. The diagrammer pane also indicates remote developer locations through special markers and tool-tips.
- Instant messaging and an audio chat channel.
- All relevant aspects of the user interface have been designed to accommodate the constantly changing state of each developer's display.
- A source code control system has been integrated to allow a CAISE project to access a central code repository.
- Build and run facilities, including protection from crosstalk when attempting to compile during times of high development activity
- Event-based collaborative feedback information, such as Degree of Interest (DOI) reports relating to other user locations within the project, and model change impact reports as the project evolves.
- A collaborative User Tree that provides a model-based view of developer's locations.

The majority of the features built in to the above tools, such as the Artifacts Pane and User Tree, are stand-alone components made available from the CAISE client widgets library, as presented in Section 3.3. The remaining collaborative features, such as multi-user editors and a UML class diagramming pane, have been implemented manually, but rely on the services of the CAISE Tool API to implement functions such as tool synchronisation and the shared modification of artifacts.

To implement tele-cursors within the Java code editor, each instance of the editor simply listens for changes in remote user cursor locations and redraws each remote cursor onto the text pane using opaque graphics rendering. To provide more advanced Groupware features, components of the Maui toolkit could be integrated with any CAISE-based CSE tool.

To implement a relaxed-WYSIWIS view within the UML class diagrammer, each time a component such as a class or interface is dragged, the drag action is captured and sent to the CAISE server via the `fireToolEvent()` API method. The UML diagrammer Tool Manager module responds to this event by adjusting its mapping of components and coordinates, and then broadcasts the adjustment event out to all tools registered for this event, which in turn update their local view of the model.

There are numerous other features that have been built into other tools such as code-age highlighting, as well as stand alone graphical components such as real-time visualisations of user activity. These features have been presented previously (Cook et al. 2004) (Cook & Churcher 2003), where the primary focus was on describing the CAISE infrastructure.

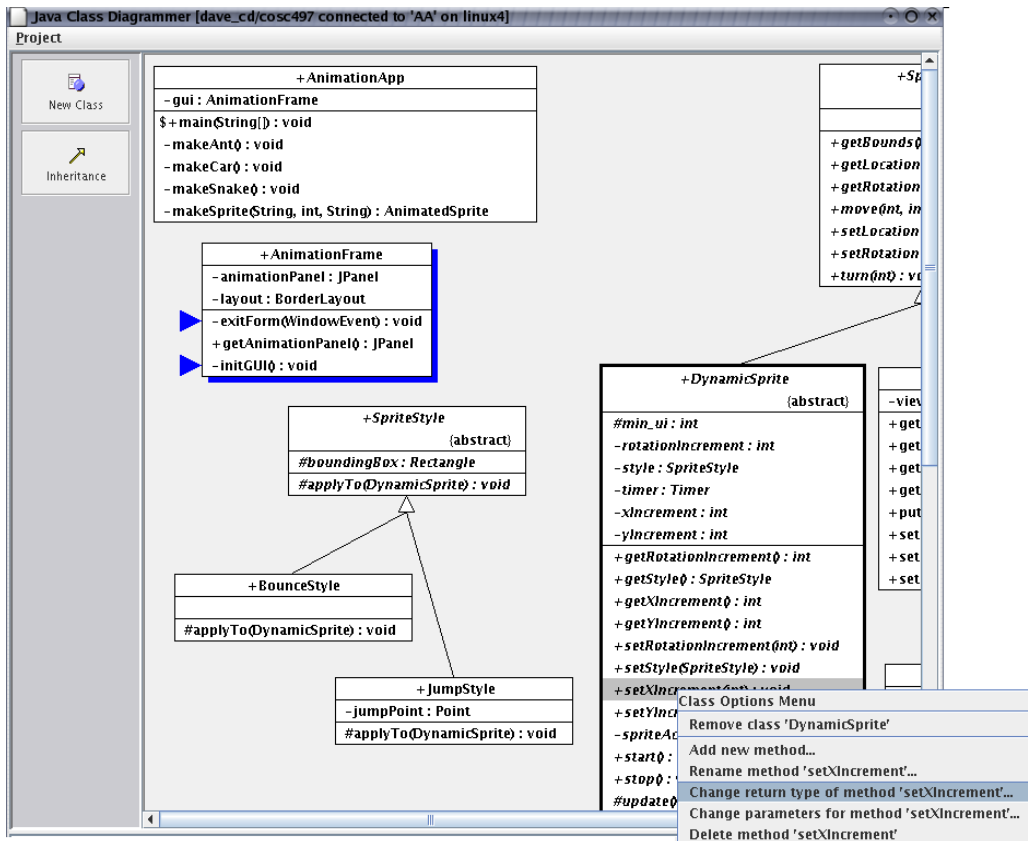


Figure 8: A UML class diagrammer. Remote user positions are indicated by the blue markers.

4.1 Server Applications

The most common type of CAISE-based tools are those that allow the direct editing, building and inspection of collaborative software projects. Other more static types of tools can be envisaged, however, such as visualisation generators and metrics querying tools. For this class of tool, the CAISE architecture supports *server-based* applications. These applications run within the server process itself, providing fast and efficient access to the software project, its artifacts and underlying semantic model.

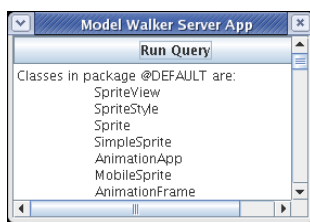


Figure 9: A server application which inspects the semantic model of a CAISE-based software project.

An example of a simple server application is presented in Figure 9. This is a trivial example, where the application simply walks a project model and displays the names of all classes within the package structure. The following code segment, however, shows how simple it is for the above application to walk the model programmatically through the use of the architecture's *model visitor*. Server applications can modify the model directly as well, causing model change events to be issued to all listening tools, which will in turn adjust their local views accordingly.

```
public class SimpleModelWalker extends CAISEServerApp {
    // called once server is ready
```

```
public void run() { initGui(); }

// called upon events
public void update(Collection events) { /* do nothing */ }

/* InitGui() method omitted */

public void jButton1ActionPerformed(ActionEvent e) {
    // get the given project
    Project project = Engine.getEngine().getProject("AA");

    // get the default package from the project's model
    PackageDecl pkg = project.getModel().getDefaultPackage();

    // print out header
    setText("Classes in package " + pkg.getSimpleName());

    // create an instance of a subclassed model visitor
    new ModelVisitorAdapter(pkg) {
        // override the visit ClassType routine
        public void visitClass(ClassType classType) {
            // write the class name out to the text panel
            addText("\n\t" + classType.getSimpleName());
        }
    }.visit(); // fire up the adapter
}
```

5 Construction of New Tools

This section describes how the tool developer can construct new CSE tools using the CAISE framework. The code segments presented in this section are taken from the Java code editor, which was discussed in the previous section and presented in Figure 7.

5.1 Tool Initialisation

Each instance of a CAISE-based CSE tool needs to establish a connection to the CAISE server. The most

appropriate time to do this is at program startup. Within CAISE, each client has a unique name, and this is given during the call to establish a server connection. The following code segment demonstrates connecting to a named CAISE server, registering an application for CAISE events, and opening an existing project.

```
// create a new instance of a CAISE handler
CAISEHandler handler = new CAISEHandler(clientName,
    serverName, TextEditor.ID);

// tell the server to notify this class of any events
handler.attachCAISECallback(this);

// open the initial CAISE project with no event filtering
handler.openProject(projectName, CAISEEvent.ALL_EVENTS);
```

5.1.1 Adding CAISE Widgets to a Tool

CAISE widgets may be constructed and added as components within a user interface in the same manner as any other standard graphics widget. The only requirement is that events from the CAISE server are passed from the containing application to each widget. The event handling code segment is given in Section 5.4.

5.2 Catching Local Tool Actions

The CAISE Tool Protocol stipulates that tools pass artifact modification events to the CAISE server, instead of allowing the tool's view of an artifact to be modified directly. To do so, tools must catch all artifact modification actions and queue them for subsequent proxying to the server. Only once the event has been processed by the server and a response has been broadcast to all tools will the local text pane be updated, as illustrated in Section 5.5.

In the following code segment, the key-presses destined for the text pane within the Java editor are captured and queued. The current cursor location is not recorded within the keystroke event—the server maintains the authoritative record of user positions to ensure consistency between all the tools, and already knows the user location at the time of the pending key press. To maintain the record of user locations, cursor location changes are another type of CAISE event governed by the CAISE Tool Protocol.

```
public void keyTyped(KeyEvent e) {

    // kill it before it gets to the editor
    e.consume();

    // ignore any keystrokes that involve alt or ctrl
    if ( (e.getModifiers() & (e.ALT_MASK|e.CTRL_MASK)) )
        return;

    // ignore escape key
    if (e.getKeyChar() == (char)27)
        return;

    // add regular key event to client queue
    enqueueEvent(new EventWrapper(e, fileName));

    // update the state of the undo menu item
    EditorFrame.this.undoItem.setEnabled(true);
}
```

5.3 Sending Tool Actions to the Server

As described in Section 3.5, the GUI thread is only responsible for capturing and enqueueing user input, and updating the local view of artifacts. The role of the worker thread is to take events from the local event queue and deliver them to the server as API method calls. As illustrated in the following code segment, the worker thread blocks until a corresponding event

has been broadcast by the server before processing any remaining queued events.

```
final class EventHandler implements Runnable {

    public void run() {
        while (isThreadRunning()) {

            // remove event from queue and pass to server
            EventWrapper ew = clientInputEvents.take();

            if (ew.event instanceof KeyEvent)
                // send key events as buffer append requests
                handler.appendSourceCodeBuffer(ew.fileName,
                    ew.event);

            // wait until the server has replied
            serverFeedbackEvents.take();
        }
    }
}
```

5.4 Listening for Server Responses

The CAISE server broadcasts events out to all registered listeners upon any significant event such as an artifact modification or a change in the project's underlying semantic model. If a tool has issued a request to modify an artifact, the server will perform the modification on its master copy and then broadcast a corresponding event to all tools. The tool that issued the request will be expecting a subsequent modification event, and all other tools are also required to adjust their local artifact views upon event notification.

The following code segment illustrates the main event loop within the Java text editor, which is representative of typical CAISE-based tools. As the editor also employs the User Tree widget, events are relayed to the widget, allowing it to update its own view of the project. The text editor also needs to keep track of user location changes in the same manner as it monitors artifact modification events, but for the sake of simplicity, this has been omitted from this example.

```
public void update(Collection events) {

    // for each event
    for (Iterator i = events.iterator(); i.hasNext(); ) {
        CAISEEvent event = (CAISEEvent)i.next();

        // inspect the event type
        switch (event.getType()) {

            // if an artifact event
            case CAISEEvent.ARTIFACT_EVENT:

                // if the artifact has been edited by anyone
                if (event.getSubType() == ARTIFACT_APPENDED)

                    // if this is the current artifact
                    if (event.getSourceEntity().equals(fileName))

                        // append the buffer of the underlying file
                        appendBufferFromRemoteChange(
                            event.getSourceUser(),
                            ((KeyEvent)event.getData()[0]),
                            ((Integer)event.getData()[1]).intValue(),
                            ((Integer)event.getData()[2]).intValue());
                    }

                // update user tree
                userTree.updateTree(event);
            }
        }
    }
}
```

5.5 Updating the Local Artifact View

To complete the MVC pattern within the CAISE event model, the final task for CSE tools upon receiving an event is to update their local view. Within the text editor, this involves appending and redisplaying the text pane upon artifact modification events.

For user location change events, this involves updating the local mapping of users and file positions and redisplaying all cursors. As the Java code editor uses a multi-user text component, artifact modification events only need to be relayed to the text pane—the multi-user component will perform the text insertion and remote modification highlighting internally.

It is important to note that each tool’s view runs no possibility of losing synchronisation with other tools or the CAISE server, barring catastrophic network failure. As long as events are captured and delivered in order to the server, and the underlying artifact is only updated within each tool in response to server events, then synchronisation is guaranteed.

```
private void appendBufferFromRemoteChange(Client editor,
                                           KeyEvent change,
                                           int positionHint,
                                           int previousFileSize) {

    // check that our user location is in sync with the server
    assertUserLocation(editor, positionHint);

    // check that the file size is in sync with the server
    assertFileSize(previousFileSize);

    // update buffer
    buffer.appendDocument(change.getKeyChar(), positionHint,
                          editor, handler.getClient());

    // restore any previously selected text
    redrawSelection(editor.equals(handler.getClient()));

    // tell auto-save timer to restart
    setBufferDirty(true);

    // yeild lock if this edit originated from this app
    if (editor.equals(handler.getClient()))
        serverFeedbackEvents.put(new Object());
}

```

6 Performance Analysis

A final consideration when discussing the design and use of collaborative tools for software engineering is that of performance. The performance of the tools must be satisfactory, and there should be no theoretical limitations of the architecture that will prevent the tools from being useful in realistic environments. While the core response speeds and resource usage of CAISE and its supporting tools have proved acceptable over a long period of subjective testing and user evaluations, it is important to note the effects of code size and number of concurrent developers on server memory load and tool response times.

6.1 Memory Load

To provide features such as code modification impact reports and DOI feedback, the CAISE server maintains a semantic model of the software within the project. An immediate concern is that of memory usage; if a large amount of memory is required for each line of code added to the model, projects of a realistically large size might be beyond the scope of the CAISE architecture.

Figure 10 presents the amount of memory used per line of code across a range of CAISE projects. For any CAISE based project, the server first loads in all packages, classes, interfaces and methods directly accessible from any Java source file. This brings the initial project model size up to around 60 MB. From that point onwards, however, most of the components that the modelled software rely upon are now loaded, and the project model size increases only linearly in relation to the number of classes and methods declared in each source file. After taking the project

initialisation into account, each line of code requires approximately one kilobyte of server memory.

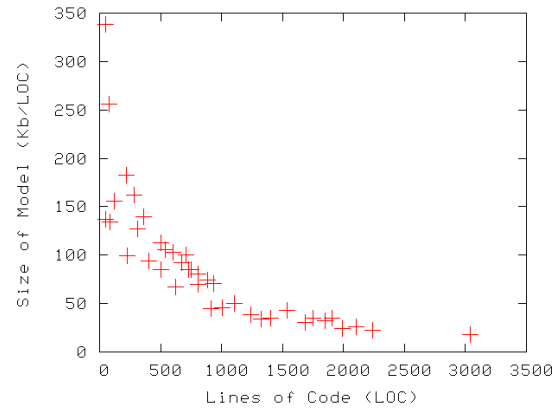


Figure 10: Lines of Code vs. Server Memory Usage.

For large software projects where there can be potentially millions of lines of code within a single revision, an alternative to an in-memory model might be required. In commercial settings, it is likely that specialised hardware can support multiple gigabytes of memory. In other situations where mass memory capabilities are not available, the CAISE architecture can easily be extended to incorporate an object-oriented database for models of potentially any size.

While the memory requirements for a CAISE-based project may seem significant, it is important to note that no other demands are placed on memory resources throughout the entire development environment. Unlike other architectures including IDEs, each CSE tool can rely on the CAISE server for all parsing, analysing and modelling of the software; tools themselves do not need to store a replica model.

6.2 Network Load

The design of the architecture ensures that network loads are as low as possible, and recent analysis of traffic verifies that for small user groups, no considerable strain is placed on a 100 Mbps Ethernet local area network. Even as the number of concurrent users increases to that of large development teams, today’s networks are capable of accommodating the load.

When testing on wide area networks, the data throughput requirements are low enough for clients to be connected to the server from dial-up networks, but the latency can cause edit delays of up to several seconds. To support low speed wide area network connections, an alternative distributed system might be necessary where the anticipated results of modification requests are immediately displayed in the originating tool’s display. In this case, a synchronisation routine will be required to run in a separate thread to resolve any modification discrepancies between tools.

At present, fault tolerance within CAISE has not been addressed. User trials and experimentation has been limited to within local networks, where error rates are low and are readily addressed by underlying communication protocols. For high-latency, high error network configurations such as intercontinental real-time software development, techniques for fault tolerance may need to be identified.

6.3 Response Times, Users and Model Size

From performance analysis, we are confident that as the number of users and the size of the project model

increases, response times will remain stable. The direct impact of increased numbers of concurrent users within a CAISE project has been observed to be negligible; the number of connected users or opened files does not have a noticeable effect on server memory usage or response time. If all users are highly active at the same time the server response times will slow down, but in reality this is a very unlikely scenario.

Even if a given project has a very large semantic model, this does not necessarily affect the response times of the server. Most operations such as adding a new method to a class or querying the model for a specific relationship only require the traversal of a fixed subset of the entire model space. Therefore, even as the model grows in size, the response times should stay approximately constant.

7 Summary

Real-time support for CSE is an important emerging field of research. The size and complexity of today's software projects far exceeds the ability of conventional single-user tools to provide an environment of fine-grained communication between developers. While source code repository systems provide a degree of control over constantly evolving software projects, there is both the demand and enabling technology for more comprehensive tool support.

To date, the large development cost in constructing new CSE tools has been a major obstacle within the field of research. In this paper, we have shown how new CSE tools can be developed rapidly within the CAISE architecture.

We have presented example CAISE-based tools, discussed the underlying protocol that ensures tool synchronisation, and have illustrated how existing multi-user widgets can be utilised within existing software engineering tools. We have also discussed the CAISE Tool API at a level of detail suitable to provide insight for potential tool developers. Finally, we have addressed various performance issues and have reasoned why CAISE-based CSE tools are able to operate satisfactorily under a range of workloads.

The CAISE architecture and associated tools performed well during recent empirical and anecdotal evaluations. After illustrating the construction and use of CAISE-based CSE tools in this paper, it is hoped that others are encouraged to develop similar tools to support the emerging field of real-time CSE.

References

- Carasik, R. P. & Grantham, C. E. (1988), A Case Study of CSCW in a Dispersed Organization, *in* 'CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems', ACM Press, New York, NY, USA, pp. 61–66.
- Churcher, N. & Cerecke, C. (1996), GroupCRC: Exploring CSCW Support for Software Engineering, *in* 'Proceedings of the 4th Australasian Conference on Computer-Human Interaction', IEEE Computer Society Press, Hamilton, New Zealand.
- Cook, C. (2005), Towards Computer-Supported Collaborative Software Engineering, PhD thesis, University of Canterbury, Christchurch, New Zealand. Work in Progress.
- Cook, C. & Churcher, N. (2003), An Extensible Framework for Collaborative Software Engineering, *in* D. Azada, ed., 'Proceedings of the Tenth Asia-Pacific Software Engineering Conference', IEEE Computer Society, Chiang Mai, Thailand, pp. 290–299.
- Cook, C. & Churcher, N. (2005a), A User Evaluation of Synchronous Collaborative Software Engineering Tools, Technical Report TR-COSC 04/05, Department of Computer Science, University of Canterbury, Christchurch, New Zealand.
- Cook, C. & Churcher, N. (2005b), Modelling and Measuring Collaborative Software Engineering, *in* V. Estivill-Castro, ed., 'Proceedings of ACSC2005: Twenty-Eighth Australasian Computer Science Conference', Vol. 38 of *Conferences in Research and Practice in Information Technology*, ACS, Newcastle, Australia, pp. 267–277.
- Cook, C., Churcher, N. & Irwin, W. (2004), Towards Synchronous Collaborative Software Engineering, *in* 'Proceedings of the Eleventh Asia-Pacific Software Engineering Conference', IEEE Computer Society, Busan, Korea, pp. 230–239.
- Cox, D. & Greenberg, S. (2000), Supporting Collaborative Interpretation in Distributed Groupware, *in* 'Proceedings of the ACM Conference on Computer Supported Cooperative Work', ACM Press, Philadelphia, PA, pp. 289–298.
- Froehlich, J. & Dourish, P. (2004), Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams, *in* '6th International Conference on Software Engineering (ICSE'04)', IEEE, Edinburgh, Scotland, United Kingdom, pp. 387–396.
- Graham, N., Stewart, H., Ryman, A., Kopae, R. & Rasouli, R. (1999), A World-Wide-Web Architecture for Collaborative Software Design, *in* 'Software Technology and Engineering Practice', IEEE, Pittsburgh, Pennsylvania, pp. 22–32.
- Greenberg, S. (1989), The 1988 Conference on Computer-Supported Cooperative Work: Trip Report, *in* 'SIGCHI Bulletin', Vol. 20 of 5, ACM, pp. 49–55. Also published in *Canadian Artificial Intelligence*, 19, April 1989.
- Hill, J. & Gutwin, C. (2003), Awareness Support in a Groupware Widget Toolkit, *in* 'Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work', ACM Press, Sanibel Island, Florida, USA, pp. 256–267.
- Lewis, S. (2005), 'Eclipse Communication Framework', Internet Homepage. <http://www.eclipse.org/ecf/goals.html>
- Reeves, M. & Zhu, J. (2004), Moomba A Collaborative Environment for Supporting Distributed Extreme Programming in Global Software Development, *in* J. Eckstein & H. Baumeister, eds, 'Lecture Notes in Computer Science', Vol. 3092, Springer-Verlag, pp. 38–50.
- Roseman, M. & Greenberg, S. (1996), 'Building Real Time Groupware with GroupKit, A Groupware Toolkit', *ACM Transactions on Computer-Human Interaction* 3(1), 66–106.
- Sarma, A. & van der Hoek, A. (2002), Palantir: Coordinating Distributed Workspaces, *in* '26th Annual International Computer Software and Applications Conference', IEEE, Oxford, England.
- Schummer, T. (2001), Lost and Found in Software Space, *in* '34th Annual Hawaii International Conference on System Sciences', IEEE Computer Society, Maui, Hawaii.