Modelling and Measuring Collaborative Software Engineering

Carl Cook Neville Churcher

Technial Report TR-05/04, September 2004
Software Engineering & Visualisation Group,
Department of Computer Science and Software Engineering,
University of Canterbury, Private Bag 4800,
Christchurch, New Zealand
{carl, neville}@cosc.canterbury.ac.nz

Abstract

Collaborative Software Engineering (CSE) supports the fine-grained real-time development of software by teams of developers located anywhere on the Internet. In this paper we describe CAISE, our CSE environment, and explore the ways in which such environments can benefit developers. We consider the rôles of heuristic evaluation, log analysis and visualisation in quantifying the benefits of CSE.

1 Introduction

Modern software engineering inevitably involves teams of developers collaboratively working on software artifacts. Challenges to success include both the size (millions of lines of code, thousands of classes) and complexity of the software under development.

However, increasingly, individual developers may be physically separated—perhaps to the extent that they are in different time zones. Similarly, the software artifacts may also be arbitrarily distributed/replicated at the developer's locations. Supporting effective collaboration between software engineers is itself a difficult problem: supporting effective collaboration between physically separated software engineers remains an open problem.

In this paper we consider Collaborative Software Engineering (CSE), which we define loosely as seamless, fine-grained, real-time collaboration between distributed software engineers who may be located anywhere on the Internet. In practice, this involves the development of tools, techniques and environments which minimise the adverse effects of collaboration with remote colleagues.

Software engineering involves teamwork and communication of many kinds. Specific examples include:

- In agile processes, such as eXtreme Programming, *pair programming* requires very close collaboration focussed on the same artifact. In CSE, the pair members need not be spatially co-located.
- Development activities such as analysis, design, testing and coding may be carried out by different combinations of individuals. CSE-mediated discussions are potentially a valuable way for effective communication and feedback between and within these groups.
- When correcting defects, team members may consult former team members, currently assigned to other projects, in order to determine the rationale for some design feature which has subsequently been identified as problematic.
- Refactoring often involves relatively minor but widespread changes. Users who are kept informed of such activity are able to avoid potential conflicts.

In our previous work (Cook & Churcher 2003, Cook, Irwin & Churcher 2004) we have focussed on the development of Caise, an infrastructure for supporting collaborative software engineering. Underlying this work is the premise that we can draw on the work of the Computer-Supported Collaborative Work (CSCW) community, which has enabled the development of collaborative approaches in other fields. However, CSE differs from typical CSCW applications, such as shared whiteboards, in several significant ways including the following:

- Artifacts produced have longer lifetimes.
- System integrity is more important.
- The cost of repairing inconsistencies is higher.
- The number, size and complexity of artifacts is higher.

Some work has been done to measure and visualise collaborative development at the granularity of version control systems such as CVS. One interesting example is Palantír (Sarma & van der Hoek 2002, Sarma, Noroozi & van der Hoek 2003) which provides a number of visualisations which help distributed developers to coordinate their efforts over the coarse-grained timescales associated with version control systems. We follow similar principles in the development of visualisations, over a range of granularities of artifacts and timescales, to support CSE.

Twidale and Nichols (Twidale & Nichols 2004) have considered support for discussions of usability issues in open source development over similar timescales. This could be supported in real time in a CSE environment.

In this paper we consider the issues associated with modelling CSE activities, assessing CSE systems in order to provide empirically-grounded development and the analysis & visualisation of CSE activity. We illustrate our arguments with examples drawn from our Caise system.

The remainder of this paper is structured as follows. In the next section, we discuss the relationships between CSE and CSCW systems. Our Caise system is presented in Section 3 and the ways in which it supports CSE are outlined. Section 4 contains an analysis of the major activities, relationships and collaborations which are evident in CSE. In Section 5 we develop a set of heuristics which will enable the capabilities and effectiveness of CSE implementations to be assessed. We show in Section 6 that effective studies of both CSE and CSE systems require detailed longitudinal studies which need to be supported by log data. We illustrate our arguments with examples from Caise applications. Finally, we present our conclusions and indicate the directions our ongoing research is taking.

2 CSE and CSCW

One approach to the problem of supporting distributed software engineering has been the development of configuration management tools (Conradi & Westfechtel 1998) and these are routinely supported by modern IDEs. The advent of eXtreme Programming (XP) (Beck 2001) and other agile software development processes relies heavily on such tools. CVS in particular has enabled the proliferation of web-based open source software development projects (over 86,000 are registered at SourceForge alone (SourceForge 2004, Bar & Fogel 2003)).

However, although they have proved extremely valuable, tools such these can only partially support CSE. Typically, the price paid by developers for coordination of their efforts is acceptance of extreme models of concurrent activity.

A *pessimistic* model uses locking to prevent shared access to artifacts: this minimises parallel development. The underlying assumptions are effectively those of a typical database application—that multiple users (transactions) concurrently accessing the same artifact are likely to lead to integrity-threatening conflicts.

By contrast, an *optimistic* model allows users to perform independent updates to the same artifact. It concentrates on providing support for resolving potential conflicts when updates are merged: this maximises parallel development but risks conflicting changes which can be difficult to resolve.

Both pessimistic and optimistic approaches are found in typical configuration management tools. Clearly, tools such as CVS have been very successful thus far. However, we contend that neither extreme is ideally suited to fine-grained real-time collaborative software engineering. Pessimistic techniques explicitly force users (transactions) to be independent (i.e. unaware of each other)—far from ideal in a team development context; optimistic techniques can lead to unpleasant surprises when integration is eventually performed.

Neither extreme is particularly suitable for CSE. Individual changes, such as adding a parameter to a method, may touch significant fractions of the whole project. If users have to wait to check out all the artifacts required (which may not necessarily be known in advance) then productivity is lowered. On the other hand, some changes may be complex and involve modifications in several places. If an optimistic approach requires that these be backed out because of conflicts then it may not be easy to reproduce them.

CSCW, also known as groupware, is a generic term for software which allows users to collaborate to perform tasks or develop artifacts (Ellis, Gibbs & Rein 1991, for example). Typical examples are shared whiteboards or text editors which allow users at different Internet nodes to make changes and to be aware of the locations and actions of other users.

As well as specific applications, a number of toolkits are available to support the development of suites of groupware applications. One notable example is Groupkit, developed at the University of Calgary's GroupLab (Roseman & Greenberg 1996).

Central to this is the concept of floor control (Dommel & Garcia-Luna-Aceves 1997, Munson & Dewan 1996). Floor control policies mediate the interleaved actions of concurrent users. While these may be as restrictive as a pessimistic locking scheme such as 2-phase locking, such policies are relatively rare. Less restrictive policies include turn-taking, where users must obtain some token or permission before they can make updates to shared artifacts.

However, the most usual policies are essentially a free-for-all, permitting users to make potentially conflicting updates at will. The underlying assumption is that *social protocols* will ensure that group members will consult one another before making potentially harmful changes. In a typical CSCW environment, users can interact via audio (telephone conference or VOIP), video and text chat facilities as well as via the groupware applications.

In order for such approaches to be successful it is necessary for users to be made aware of the locations, actions and intentions of others. Groupkit, for example, provides a variety of awareness mechanisms such as multiple cursors (telepointers) and multi-user scroll bars.

The applicability of CSCW techniques to software engineering has been demonstrated for some specific contexts (Churcher & Cerecke 1996, Schummer 2001, for example). Early attempts typically used domain-independent CSCW components. While this naïve approach was sufficient to cope with small, simple systems (Greenberg 1989), the technology did not scale well to larger, more complex situations (Grudin 1992, Grudin 1994) and CSCW has largely been abandoned in such contexts.

However, our approach, which layers CSCW concepts onto a core IDE engine, is robust and potentially much more appropriate for CSE. Caise maintains a single copy of all artifacts, manages syntactic and semantic models, detects and propagates events to users and performs a variety of other functions. Client tools are supported by a server that fully understands the components they are manipulating and the relationships of the underlying software model.

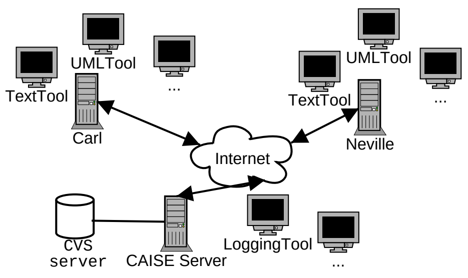


Figure 1: CSE

Figure 1 illustrates the high level architecture of our system. Individual users, Carl and Neville, are located at different Internet nodes and each has a set of development tools such as code editors and diagramming tools. These individual tools inform the CAISE server of updates they make to project artifacts. In return, the server ensures that their local copies of artifacts are refreshed to reflect changes made by other users as they occur.

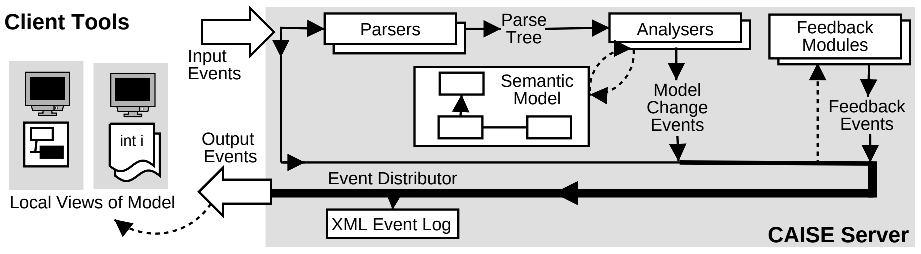


Figure 2: CAISE architecture

3 The CAISE architecture

The client/server architecture of CAISE is presented in Figure 2 and supports the CSE activities shown schematically in Figure 1. It comprises four core entities: CAISE server, client tools, event propagation and semantic model.

CAISE **server** The server is responsible for managing all users within the software project and their associated software artifacts. It performs many of the functions of an IDE: maintaining a cache of artifacts, constructing a symbol table, keeping track of references between units of code, and supporting multiple languages and tools. However, the CAISE server differs from an IDE in some significant ways: it is shared simultaneously between multiple users and tools, allows only one 'live' version of artifacts, supports collaboration as a core function and shares its full semantic knowledge with client tools.

Central to the server is the *semantic model of software.* Whilst each artifact is managed and stored by the server, the model stores the corresponding project semantics: information such as the symbol table, references between symbols and the list of currently unresolved references.

Language-specific *parsers* transform source code into a representations more suitable for exchange and further processing. Upon source file modification by a client tool, the server generates a corresponding *parse tree* in order to update its semantic model and inform users. The parse tree is also delivered to interested client tools, allowing them to update their local models and project views.

The server uses language-specific *analysers* to process the updated parse trees, generating corresponding semantic model updates. Analysers understand both the structure of the parse tree and the semantic model, and perform the task of updating the model to reflect the difference between the old and new parse trees.

User-defined *feedback modules* are invoked by the server following changes to the semantic model, or user events that suggests a client has changed location in an artifact. Feedback modules inspect the current state of the model and generate client-specific messages for tools to respond to.

All user events (such as a change of location in an artifact), change events, feedback events and user communication events (such as chat messages) are collated by the server. Whilst these events, where relevant, are distributed to client tools, the server also appends them to the *event log*. The event log is available to clients and can be used for a variety of purposes, such as playback of the project development, inspection of specific events, mining for patterns of usage, and metrics gathering. Some examples are given in Section 6.

In addition to the set of events, the event log also stores data about all project users and tools, together with all declared symbols. This data augments the individual event data contained in the log, providing the event log viewer with a considerable amount of information related to the editing of artifacts and the resultant changes to the semantic model.

Client tools Caise clients are software engineering tools that conform to the Caise client API. Each client tool has a local model—in the case of a text editor, this will typically be the set of source file artifacts in the software project. When remote changes occur, the tools are notified by the Caise server and in turn update their own models and corresponding views. For example, a UML class diagramming tool could have a local model that takes into account only the classes belonging to a particular set of packages. This tool would only update its local model when remote updates to specific classes occur.

Event propagation The actions of client tools and the reactions of the Caise server are bound together by the event model. When tools create, open or modify an artifact, *input* events are generated and delivered to the server. The server then responds to the events, typically by updating the semantic model—at which point *change* events may be generated. Finally, user-defined *feedback* events may also be generated in response to specific change and input events. Once all events have been collated, they are delivered to the appropriate listeners such as code editors and real-time visualisation/project management tools.

Semantic model There are several sources of information within the Caise architecture. The authoritative state of the software project is represented by the semantic model, which has been described elsewhere (Irwin & Churcher 2003)). This includes the complete syntactic structure of the projects, represented by the current set of parse trees.

Parse trees hold the explicit structure of all declarations and statements within a software project. Additionally, modification information for each element is stored in the parse tree. For example, if a parse tree contains (amongst other things) a method declaration, it is possible to inspect the creation date of the method, plus any semantic change information such as a changed return type or an addition of a local variable.

An API exists to inspect and append the semantic model. A visualisation tool, for example, might inspect the model to generate a visualisation of the project's current class structure. Similarly, a project management tool might inspect the model periodically to produce a metrics report.

In a more reactive manner, event listeners can be instantiated within client tools, which respond to given events. For example, a real-time project management tool might listen to model change events and raise a warning when a class appears to be in a state of conflict between several developers.

To view the full history of the project, the event log may be inspected. The declared symbols extracted from the event log may then be looked up in the semantic model if required, providing vast information about the symbol declarations and related components.

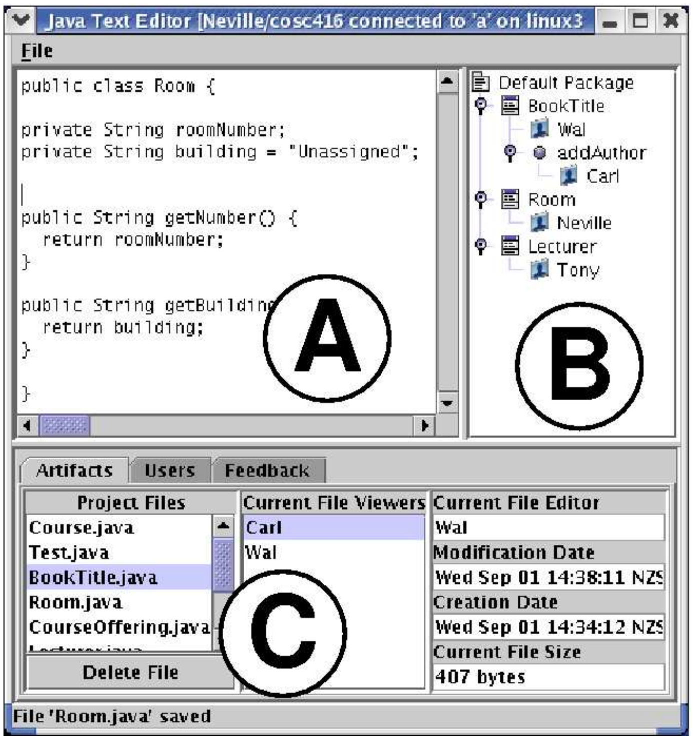
3.1 The Java editor tool

An example client tool within the Caise architecture presented in Figure 3. This tool represents a Java code editor currently opened by user 'Neville'. This editor allows any number of participating users to construct and maintain a Java-based program collaboratively and in real-time. Three main components are identified within this tool:

Editor panel: (labeled 'A' in Figure 3) a synchronous groupware widget that allows multiple distributed users to edit a given buffer at the same time. As the Caise architecture supports the propagation of user input events to interested listeners in real-time, all editors maintain identical copies of the artifact buffer.

User tree: ('B') provides context awareness for each tool user, and is particularly useful within the editor. Whilst the editor only knows how to display and support the modification of text files, the user tree provides a dynamic model-centric view of the software project—independent of the actual source files being edited.

Client panel: ('C') provides information related to the artifacts within the software project, such as the artifact editors and viewers, file sizes and modification dates. The client panel also provides an area for the display



```
public class Room {

private String roomNumber;
private String building = "Unassigned";

|
public String getNumber() {
  return roomNumber;
}

public String getBuildin
  return building;
}

}
```

Project Files
Course.java
Test.java
BookTitle.java
Room.java
CourseOffering.java

Current File Viewers
Carl
Wal

Current File Editor
Wal
Modification Date
Wed Sep 01 14:38:11 NZS
Creation Date
Wed Sep 01 14:34:12 NZS
Current File Size
407 bytes

File 'Room.java' saved

Figure 3: A Caise session

of customised feedback messages from the server. It is also used to send chat and voice messages to other users.

The three components listed above are part of the Caise client widgets package; all such widgets can be used within any Java application.

More detailed information related to the Caise architecture and associated tools is available elsewhere (Cook & Churcher 2003, Cook et al. 2004).

4 Understanding CSE

In order to assess both the benefits of CSE in general and the quality of the Caise implementation in particular, several distinct aspects of systems such as Caise need to be considered.

Architecture: Caise employs a client-server architecture whereas replicated systems are more common in conventional CSCW applications. Although some research has been done on principles for evaluating architectures (Tvedt, Costa & Lindvall 2004, for example), this aspect is not currently a primary concern for us. The major factor in determining our architecture choice is the need to be able to support the throughput needed in order to propagate and coordinate user updates to artifacts.

Infrastructure: A CSE environment should provide a means of adding new tools, languages and server facilities. Caise provides an extensible framework for both client tools (e.g. adding a new kind of diagramming tool) and server applications (e.g. responding to the changing semantic model). In addition, a set of widgets such as the user tree is available.

Usability: Individual tools, such as editors and diagrammers, must have sufficient features that professional developers will be prepared to use them. Both taskwork-oriented features, such as formatting and searching, and teamwork-oriented features, such as awareness indicators, must be available.

In order to determine the most effective ways to use CSE we need to develop an understanding of what actually takes place in (collaborative) software engineering.

Design patterns for software engineering (Gamma, Helm, Johnson & Vlissides 1995), based on Alexander's concepts from the architectural domain (Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King & Angel 1977, Alexander 1979) have been remarkably successful. Their success has led to efforts to assemble pattern languages for other fields. Martin and Sommerville (Martin & Sommerville 2004) have identified a number of patterns which reflect the ways in which groups of people interact to perform tasks. Examples include *Artifact as an audit trail* and *Collaboration in small groups*: further detail is available at `http://polo.lancs.patterns` and `http://polo.lanc.pointer`.

In an analogous manner, we are working to identify patterns more specific to CSE.

As a first step, we identify several interaction modes, each characterised by the degree of coordination required (teamwork) and the nature of the activity (taskwork):

Private: A user effectively withdraws from the group temporarily, typically to convince herself of the viability of a change before revealing it to the others. Such a user may require the rest of the project to appear frozen in time. Ideally, it should be possible to re-integrate the change rather than having to repeat it publicly.

Independent: Users are located in regions of code whose semantic relationships are sufficiently weak that they can safely assume independence. Frequent communication is unnecessary and project integrity is not threatened by independent updates. An example might involve user A editing a GUI (view) class to alter a menu, user B editing a customer record (model) class and user C adding a new package which does not yet interact with other classes.

Follow the leader: One user takes others on a guided tour. An example would be to show others the details of a recently completed change. Strict What You See Is What I See (WYSIWIS) might be used to coordinate views, particularly if all users are using the same tool. However, in a more relaxed scenario, users would navigate individually, guided by audio commentary and gestures.

Action/Reaction: Stronger constraints exist as users become closer in physical, logical or semantic terms. For example, if user A changes the type of a parameter in a method definition in class C1 then another user editing class C2 may need to update calls to that method. Changes made by a user (the actions) to aspects such as the number and type of class properties, the parameters and return types of methods or the inheritance and interface structure will require responses (reactions) from other users whose work is potentially affected. Awareness mechanisms can alert users to possible *threats* (e.g. another user is editing a superclass). Collaboration support mechanisms, such as text or audio channels and gestures, can then be employed to discuss and resolve the issues.

Mêlée: Several users are making (potentially-) conflicting changes to a set of artifacts and these will be in a state of flux for a period. Such changes would typically be negotiated in advance, and mediated throughout, by infrastructure features such as an audio channel.

5 Heuristics

As is the case for other kinds of groupware, and for software engineering in general, it is important to maintain a balance between development of CSE infrastructure and ongoing evaluation. If too long is spent performing detailed analyses of prototype systems then not only are results likely to be of marginal use but also the development process is likely to be delayed or misled. On the other hand, to ignore evaluation is to risk failure because development is not guided and informed by empirical work.

Evaluating systems and techniques with typical user groups on realistic problems is difficult, time consuming and expensive. Various approaches have been developed.

User trials are particularly suited to evaluating Human-Computer Interface (HCI) and usability aspects. These would be most useful towards the end of our project when a full range of industrial strength tools is available. At that point we will wish to quantify such factors as the relative merits of alternative feedback/feedthrough mechanisms and the balance between the benefits of awareness of others and the potential distractions from one's own tasks.

Field studies and case studies are long term undertakings conducted in realistic industrial environments in order to determine the domain-specific tasks which must be supported and to observe how particular systems are used in practice. As well as being expensive in terms of both time and cost they also have difficulties such as provision of control groups: consequently, they are most useful when the systems to be evaluated are at a mature level. In our case, these will be valuable to explore the patterns of collaboration amongst users and the effectiveness of individual techniques on particular categories of tasks.

In order to gain the most from costly evaluations it is important to be able to address the issues of assessing systems which are in early stages of development. This allows the developers to use results to improve the system rather than simply quantify its performance. A range of so-called 'discount' evaluation techniques have been developed to achieve this. These include heuristic evaluation (Nielsen & Molich 1990, Nielsen 1992, Nielsen & Landauer 1993), in which small groups of evaluators seek violations of a given set of heuristics. Results suggest that these techniques can be very effective in detecting faults, thereby enabling them to be corrected earlier in the development cycle.

A set of heuristics for evaluation of groupware has recently been proposed (Baker, Greenberg & Gutwin 2001, Baker, Greenberg & Gutwin 2002). We summarise them here and indicate briefly how they relate to specific CAISE features.

CSCWi *Provide the Means for Intentional and Appropriate Verbal Communication.* CAISE provides text chat and an audio channel. External systems, such as telephone conferencing and web cams, may also be used.

CSCWii *Provide the Means for Intentional and Appropriate Gestural Communication.* CAISE provides a user tree (see Figure 3) which indicates the location (scope) of each user. Individual tools may supplement this by implementing features such as telepointers.

CSCWiii *Provide Consequential Communication of an Individual's Embodiment.* Currently implemented via the user tree.

CSCWiv *Provide Consequential Communication of Shared Artifacts (i.e. Artifact Feedthrough).* Client tools' buffers remain synchronised to reflect changes to the underlying artifacts. Individual tools may implement features such as colour-coding for the age of updates.

CSCWv *Provide Protection.* The default access policy in CAISE is to rely on social protocols while clients provide features such as the ability to undo changes.

CSCWvi *Management of Tightly and Loosely-Coupled Collaboration.* The user tree and client panel enable users to assess activities of interest. Feedback, tailored to reflect the users' interests, is used to alert users to potential conflicts.

CSCWvii *Allow People to Coordinate Their Actions.* Communication channels and other feedback mechanisms support coordination.

CSCWviii *Facilitate Finding Collaborators and Establishing Contact.* The CAISE session management tools, such as the user tree, indicate which users, tools and artifacts are currently active.

It is useful to distinguish *taskwork*, task-specific actions, and *teamwork*, actions specific to group performance of tasks. Collaboration Usability Analysis (CUA) (Pinelle & Gutwin 2002, Pinelle, Gutwin & Greenberg 2003) provides a technique for modelling domain-specific tasks in order to form a basis for heuristic evaluation.

5.1 Heuristics for CSE evaluation

We see heuristic evaluation as a valuable complement to other techniques for evaluating CSE systems, particularly for infrastructure and capability assessment—the areas in which we most desire experiment-driven feedback during development.

CSEi	*Support multiple views of artifacts.* A given Java class may be represented in different ways by individual client tools such as a text editor, folding editor, user tree or UML class diagrammer. Changes made to the underlying artifact by any tool should be reflected appropriately in each view. Caise tools send updated artifacts to the server. In return, they receive syntax trees corresponding to artifacts which have been updated by others.
CSEii	*Support Degree of Interest based feedback/feedthrough.* Central to software engineering activities such as refactoring and comprehension is the notion of the neighbourhood (context) of a particular component or change (focus). The neighbourhood indicates the most relevant components to be taken into account from the viewpoint of the focus. For example, when modifying a method the neighbourhood might include the method itself, the methods it invokes & is invoked by, its host class and its parent class. This focus+context concept is familiar in visualisation. One common approach is the use of fisheye-view techniques (Furnas 1986, Sarkar & Brown 1994) to de-emphasise features not in the neighbourhood of the focus. When Caise clients update artifacts, events are generated whose foci are located at the corresponding parse tree nodes. Caise tailors feedback according to the neighbourhood of such nodes, determined by the semantic model, use preferences and specific client capabilities.
CSEiii	*Support fine-grained integrity.* CSE requires more powerful approaches than simple CSCW applications in order to reflect the semantic and syntactic structures implied by the source code or other artifacts. Caise uses parse trees as the basis for the semantic model it maintains.
CSEiv	*Support multiple physical and logical granularities.* Physical granularity levels reflect physical partitioning (URL, directory, file, line, . . .) while logical granularity levels (package, class, method, block, statement, expression, . . .) reflect syntactic structure.
CSEv	*Support deep syntactic- & semantic-based awareness & feedback.* The generic CSCW heuristics address issues such as notification of changes in the location of other users. In CSE, it is also important to be aware of changes at a semantic level (e.g. method foo() has been deleted from class Bar) or altered relationships involving components and users (e.g. another user is editing a method which the method you are editing overrides). Caise clients are notified of changes to the semantic model (including inferred relationships) and can reflect these as appropriate.
CSEvi	*Support semantic relationships.* Updates to artifacts lead to *indirect*, and often subtle, changes in semantic relationships (extends, overloads, overrides, calls, uses, . . .) which should be indicated to users.
CSEvii	*Support private work and re-integration.* Users can work against a snapshot of the project state and make experimental changes which will not be seen by others. In Caise this simply involves client tools temporarily detaching from the server.
CSEviii	*Support builds at different temporal granularities.* A rapidly evolving project, where developers make interleaved changes, could potentially spend much of its time in a broken state in which many components are unable to compile. CSE systems must accommodate artifacts that are temporarily un-parsable and projects that have unresolved code references. Caise propagates modification events to users directly accessing the same artifacts, ensuring that their views are synchronised at short timescales. The underlying semantic model is updated only when syntactic correctness is restored, so that on a coarser timescale, other users always build against a correct version.

Figure 4: CSE Heuristics

The heuristics and task modelling techniques proposed for CSCW (Baker et al. 2001, Baker et al. 2002, Pinelle & Gutwin 2002, Pinelle et al. 2003) are somewhat generic. We aim to extend these in two ways.

Firstly, we see merit in establishing additional domain specific heuristics for CSE since this differs in many ways from the typical CSCW application area.

Secondly, we advocate analysis and visualisation of Caise logs. This allows us to mimic many of the beneficial aspects of case studies.

However, we contend that logs can also reveal a great deal about collaboration patterns, system performance, task complexity and many other factors. In particular, they can indicate where refinement or extension of heuristics is appropriate, thereby improving the quality of subsequent heuristic evaluations.

Our current set of CSE-specific heuristics, to be considered alongside the generic CSCW heuristics discussed earlier, appears in Figure 4. A brief rationale for the inclusion of each heuristic is given, together with a brief indication of its relevance to the current Caise version.

Heuristic evaluation, based on the combination of both sets, leads to the identification and classification of problems and issues with CSE, CSE implementations and specific tools.

As an example, a problem identified in our current version as a violation of heuristic Cscwiii is "The user tree shows user location well, but does not indicate the transition from the previous location, making it difficult to decide what changes in location have occurred." Similarly, a problem identified in our current version as a violation of heuristic CSEv is "Semantic feedback is predominantly delivered via text messages. A metaphor more tightly coupled to the artifact representation would be more effective."

6 Visualising CSE

In this section, we consider the use of visualisations based on analysis of Caise event logs. Such visualisations can provide valuable information about such things as patterns of collaboration, user activity profiles and sequences of operations in refactoring. This both complements and informs heuristic evaluations.

Empirical data from logs helps ensure that the sets of heuristics used are valid, representative and complete. Such data guides the ongoing process of refining sets of heuristics. In return, heuristics suggest patterns which should be observable in event logs.

Caise incorporates an XML-based logging facility which records data about such things as users & tools, events resulting from user activity and the artifacts affected. Server-side tools may process the logs in real-time in order to obtain information, such as cumulative activity indicators, for propagation to users. Alternatively, logs may be processed off-line in order to perform detailed analyses.

The potential uses of (even relatively unsophisticated) visualisations in groupware have been recognised (Begole, Tang & Hill 2003). The pipeline-based techniques we have developed for software and information visualisation (Irwin & Churcher 2002, Churcher, Irwin & Cook 2004) are applicable to Caise event log visualisation.

Figure 5 shows a typical log visualisation pipeline. Firstly, XSLT or other filters select and process the required data. In subsequent stages, layout tools produce 2D or 3D visualisations which are then rendered for user exploration. These may be as varied as spreadsheet graphics or virtual worlds.

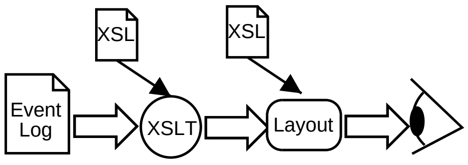
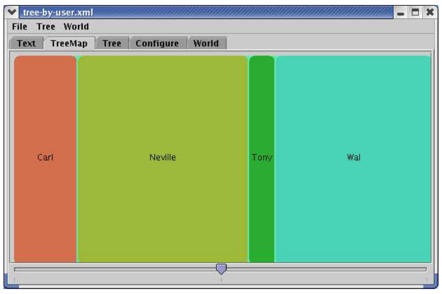
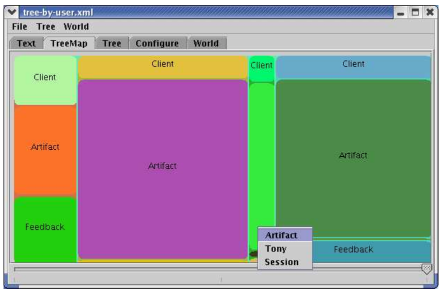


Figure 5: Visualising log data

The event logs conform to a DTD, `http://www.cosc.canterbury.ac.nz/dtd/CAISEEventLog.dtd`, enabling validation to be performed. Filters, typically implemented in XSLT, extract and format the data required for specific visualisations.



(a) Events originated by each user



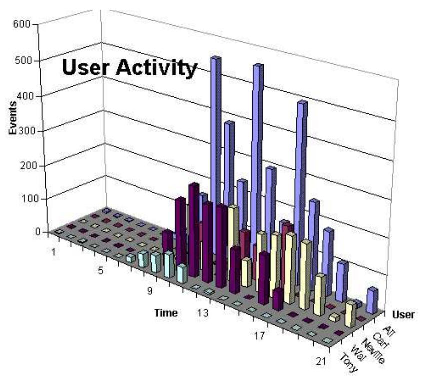
(b) Events of each type originated by each user

Figure 6: Treemaps showing events in a CAISE session

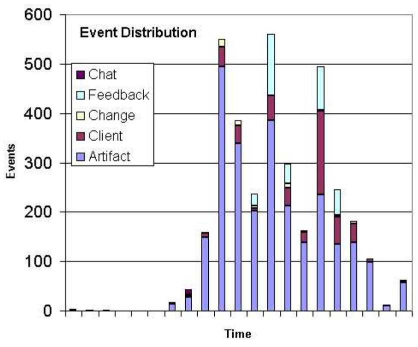
We will illustrate the visualisation process with some analyses of log data from a CAISE session of approximately 30 minutes, involving four users working on a project consisting of ten java classes.

Figure 6 shows two views of a visualisation based on treemaps (Johnson & Shneiderman 1991): in this case the pipeline ends with one of our own tools. The log is transformed into a tree representing a hierarchy of events structured thus: session → user→ event type. Similarly, other structures (e.g. component → event type → user may readily be obtained. Figure 6(a) shows that two users, Neville and Wal, are responsible for most of the events generated in the session. Figure 6(b) provides additional information about the proportion of events of each type resulting from individual users' actions. In this case, it can be seen that Carl and Wal have a greater proportion of feedback events than the other users. From this information we can deduce that Tony was the least active user in this session (in fact he left before it ended); Neville and Wal were responsible for the bulk of the coding done during the session; Carl and Wal collaborated most closely (i.e. concurrently edited the same artifacts) while Neville and Wal worked more independently.

Figure 7 illustrates some temporal analysis options for the same data set. In this case the pipeline ends with a file readable by Excel. Figure 7(a) shows the number of events generated by the activity of each user during a 100 second



(a) All events per user



(b) Events by type

Figure 7: Temporal analysis

interval as well as the overall totals. Peaks and lulls in activity can be seen clearly. In Figure 7(b) the events are broken down by type, irrespective of the user responsible for their generation. Again patterns are evident. Most events in this session are Artifact events, since significant text entry is occurring; Client events are mainly associated with location changes within artifacts; Change events arise from semantic changes such as altered inheritance relationships and are associated with Feedback events which alert other users to the changes.

Finally, Figure 8 shows the specific artifacts (Java source files) modified by each user during the session. In this case, the pipeline produces a file for the popular `dot` layout tool (Gansner & North 1999). In a dynamic version of this graph, edges are added and removed to reflect the current session state.

The visualisations presented in this section, although they contain considerable amounts of useful information, are still relatively simple. We are developing a range of more sophisticated visualisations, both for analysis purposes and for inclusion in client tools. An example is the use of colour to indicate attributes, such as time since last edit or last user to edit) lines of code in a text editor. Another example is the provision of feedback about the rate of change of components.

7 Conclusions & further work

In this paper, we have described collaborative software engineering, reported on the current state of our Caise implementation and investigated approaches for understanding and evaluating CSE applications.

We have considered several approaches to evaluation of CSCW applications in general, and CSE in particular, and have explored the benefits offered by heuristic evaluation techniques. CSE-specific heuristics have been developed in order to enhance the effectiveness of heuristic evaluation in the CSE domain.

The symbiotic relationship between heuristics and empirical data obtained from event logs has been developed. In order to achieve the goal of improving CSE implementations, it is necessary to employ heuristics which accurately reflect the processes and practices occurring in realistic CSE use. Problems identified using such heuristics are likely to directly relevant in the CSE do-

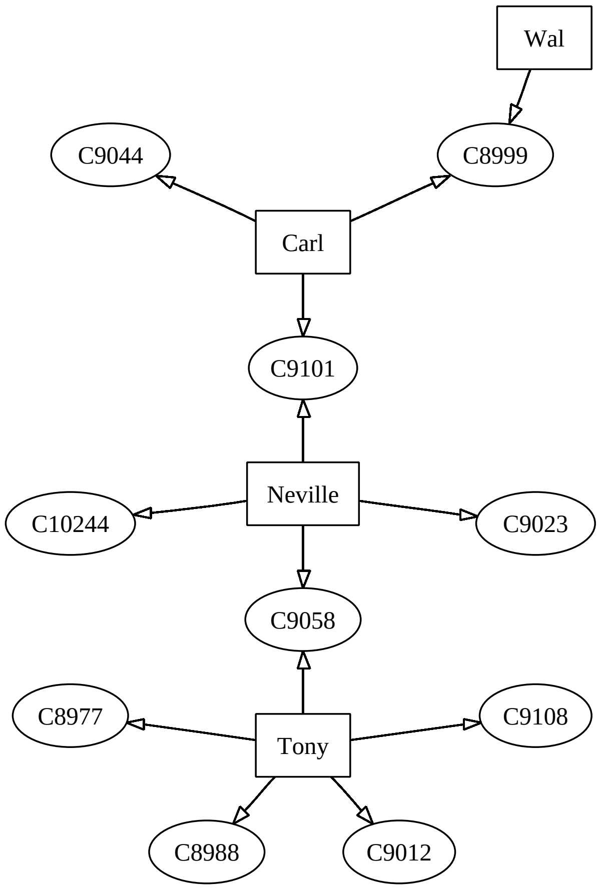


Figure 8: Artifacts accessed

main. However, not all relevant scenarios are readily accessible to heuristic evaluators. Analysis and visualisation of empirical data obtained from event logs complements heuristics, highlighting patterns and issues at various levels of detail.

As our CAISE implementation matures, we expect to make greater use of both heuristic evaluation and log visualisation to assess the state of our current version and to steer ongoing developments.

Our work on event modelling and logging is particularly helpful as we refine the APIs for development of CAISE client tools.

We believe that the increased understanding of CSE which results from our empirical work will help us, and other CSE researchers, to improve the quality of CSE infrastructure implementations. In addition, it will assist in the design of more effective client tools for supporting collaborating software engineers.

References

Alexander, C. (1979), *The Timeless Way of Building*, Oxford University Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. & Angel, S. (1977), *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press.

Baker, K., Greenberg, S. & Gutwin, C. (2001), Heuristic evaluation of groupware based on the mechanics of collaboration, *in* M. Little & L. Nigay, eds, 'Engineering for Human-Computer Interaction: Proc ECHI 2001', Vol. 2254 of *Lecture Notes in Computer Science*, Springer-Verlag, Toronto, Canada, pp. 123–139.

Baker, K., Greenberg, S. & Gutwin, C. (2002), Empirical development of a heuristic evaluation methodology for shared workspace groupware, *in* 'Proceedings of the 2002 ACM conference on Computer supported cooperative work', ACM Press, pp. 96–105.

Bar, M. & Fogel, K. (2003), *Open Source Devlopment with CVS*, 3rd edn, Paraglyph Press.

Beck, K. (2001), *Extreme Programming Explained: Embrace Change*, Addison-Wesley.

Begole, J. B., Tang, J. C. & Hill, R. (2003), Rhythm modeling, visualizations and applications, *in* 'Proceedings of the 16th annual ACM symposium on User interface software and technology', ACM Press, pp. 11–20.

Churcher, N. & Cerecke, C. (1996), groupCRC: Exploring CSCW support for software engineering, *in* J. Grundy & M. Apperley, eds, 'OZCHI'96', IEEE Press, University of Waikato, Hamilton, New Zealand, pp. 62–68.

Churcher, N., Irwin, W. & Cook, C. (2004), Inhomogeneous force-directed layout algorithms in the visualisation pipeline: From layouts to visualisations, *in* N. Churcher & C. Churcher, eds, 'InVis.au 2004: Proceedings of the Australasian Information Visualisation Symposium', Vol. 35 of *Conferences in Research and Practice in Information Technology*, ACS, Christchurch, New Zealand.

Conradi, R. & Westfechtel, B. (1998), 'Version models for software configuration management', *ACM Comput. Surv.* **30**(2), 232–282.

Cook, C. & Churcher, N. (2003), An extensible framework for collaborative software engineering, *in* D. Azada, ed., 'APSEC 2003: Proceedings of the 10th Asia-Pacific Software Engineering Conference', IEEE Press, Chiang Mai, Thailand, pp. 290–299.

Cook, C., Irwin, W. & Churcher, N. (2004), Towards synchronous collaborative software engineering, *in* 'Proc APSEC2004: 11th Asia Pacific Software Engineering Conference', Busan, Korea.

Dommel, H.-P. & Garcia-Luna-Aceves, J. J. (1997), 'Floor control for multimedia conferencing and collaboration', *Multimedia Syst.* **5**(1), 23–38.

Ellis, C., Gibbs, S. & Rein, G. (1991), 'Groupware: Some issues and experiences', *Communications of the ACM* **34**(1), 38–58.

Furnas, G. (1986), Generalised fisheye views, *in* 'Proc ACM SIGCHI '86 Conference on Human Factors in Computing Systems', pp. 16–23.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Gansner, E. R. & North, S. C. (1999), 'An open graph visualization system and its applications to software engineering', *Software—Practice and Experience* **30**(11), 1203–1233.

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.

Grudin, J. (1992), Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces, *in* D. Marca & G. Bock, eds, 'Groupware: Software for Computer-Supported Cooperative Work', IEEE Press, Los Alamitos, CA, pp. 552–560.

Grudin, J. (1994), Groupware and social dynamics: Eight challenges for developers, *in* 'Communications of the ACM', Vol. 37 of *1*, ACM Press, pp. 92–105.

Irwin, W. & Churcher, N. (2002), XML in the visualisation pipeline, *in* D. D. Feng, J. Jin, P. Eades & H. Yan, eds, 'Visualisation 2001', Vol. 11 of *Conferences in Research and Practice in Information Technology*, ACS, Sydney, Australia, pp. 59–68. Selected papers from 2001 Pan-Sydney Workshop on Visual Information Processing.

Irwin, W. & Churcher, N. (2003), Object oriented metrics: Precision tools and configurable visualisations, *in* 'METRICS2003: 9th IEEE Symposium on Software Metrics', IEEE Press, Sydney, Australia, pp. 112–123.

Johnson, B. & Shneiderman, B. (1991), Tree-maps: A space-filling approach to the visualization of hierarchical information structures, *in* G. Nielson & L. Rosenblum, eds, 'proc. Visialization '91', IEEE Computer Society Press, Los Alamitos, CA, pp. 284–291.

Martin, D. & Sommerville, I. (2004), 'Patterns of cooperative interaction: Linking ethnomethodology and design', *ACM Trans. Comput.-Hum. Interact.* **11**(1), 59–89.

Munson, J. P. & Dewan, P. (1996), A concurrency control framework for collaborative systems, *in* 'Computer Supported Cooperative Work', pp. 278–287. *citeseer.ist.psu.edu/munson96concurrency.html

Nielsen, J. (1992), Finding usability problems through heuristic evaluation, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 373–380.

Nielsen, J. & Landauer, T. K. (1993), A mathematical model of the finding of usability problems, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 206–213.

Nielsen, J. & Molich, R. (1990), Heuristic evaluation of user interfaces, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 249–256.

Pinelle, D. & Gutwin, C. (2002), Groupware walkthrough: adding context to groupware usability evaluation, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 455–462.

Pinelle, D., Gutwin, C. & Greenberg, S. (2003), 'Task analysis for groupware usability evaluation: Modeling shared-workspace tasks with the mechanics of collaboration', *ACM Trans. Comput.-Hum. Interact.* **10**(4), 281–311.

Roseman, M. & Greenberg, S. (1996), 'Building real-time groupware with GroupKit, a groupware toolkit', *ACM Trans. Computer-Human Interaction* **3**(1), 66–106.

Sarkar, M. & Brown, M. (1994), 'Graphical fisheye views', *Communications of the ACM* **37**(12), 73–84.

Sarma, A., Noroozi, Z. & van der Hoek, A. (2003), Palantír: Raising awareness among configuration management workspaces, *in* 'Proc ICSE2003: 25th International Conference on Software Engineering', Portland, Oregon, pp. 444–454.

Sarma, A. & van der Hoek, A. (2002), Palantír: Coordinating distributed workspaces, *in* 'Proc COMPSAC 2002: 26th Computer Software and Applications Conference', Oxford, England, pp. 1093–1097.

Schummer, T. (2001), Lost and Found in Software Space, *in* '34th Annual Hawaii International Conference on System Sciences', IEEE, Maui, Hawaii.

SourceForge (2004), 'SourceForge.net Home Page', `http://sourceforge.net/`.

Tvedt, R. T., Costa, P. & Lindvall, M. (2004), Evaluating software architectures, *in* M. Zelkowitz, ed., 'Architectural Issues', Vol. 61 of *Advances in Computers*, Academic Press, New York, pp. 2–43.

Twidale, M. & Nichols, D. (2004), Usability discussions in open source development, Working Paper 08/2004, Department of Computer Science, University of Waikato, Hamilton, New Zealand.