An Extensible Framework for Collaborative Software Engineering

Carl Cook and Neville Churcher
{c.cook, neville}@cosc.canterbury.ac.nz

Technial Report TR-02/03, July 2003
Department of Computer Science
University of Canterbury,
Christchurch, New Zealand

Abstract

The size, complexity and duration of typical software engineering projects means that teams of developers will work on them. However, with the exception of version control systems, the editors, diagrammers and other tools used will generally support only a single user. In this paper, we present an architecture for bringing to software engineering development environments the advantages of awareness of the presence, and the intentions and actions of others. Thus far, the applications of such facilities have been primarily in simple Computer Supported Collaborative Work (CSCW) tools, such as shared whiteboards, where the corresponding artifacts, unlike those of software engineering, are typically both simple and transient. We describe our implementation of the architecture and prototype tools and illustrate the benefits of providing support for real-time collaboration between developers located anywhere on the Internet. We also describe how our architecture, which is based on a parse tree representation of artifacts, may be extended readily to include new tools, languages, and notations or be customised to provide new awareness mechanisms.

1 Introduction

Software engineering is inherently a team-based activity; most projects today are complex and typically require several iterations of design, development, and testing by numerous engineers. Unfortunately, very few tools are available to support real-time, or synchronous, collaborative software engineering. There are many tools well suited to synchronous unstructured collaboration such as chat applications and shared digital whiteboards, but software engineers require additional support for the sharing of source files and other artifacts.

Version control systems such as CVS [2] and VA Software's SourceForge are the most popular and successful means of collaborative software development today. They are, however, fundamentally asynchronous or off-line systems; artifacts such as source code and class diagrams are locked exclusively by a single user for modification. This results in code modifications made in isolation from all other users, and the impact of code changes is not known until the entire system is rebuilt, which is typically an overnight process.

Consequently, the full advantages of spontaneous teamwork are not being utilised by software engineers at present, and the practice as a whole appears content in relying fully on asynchronous tools such as email and version control systems. We consider this as a serious obstacle to the advance of software engineering, and a problem that is likely to remain unresolved given the current lack of activity in the field of asynchronous software development.

Just over a decade ago, Cscw systems promised to revolutionise the way we worked together. Applications would be replicated to every participant, and conferencing and whiteboard tools would reduce the affects of any physical separation between coworkers, as discussed by Saul Greenberg [11]. This had an immediate appeal to the practice of software engineering—design and development tools could be converted to multiuser versions, allowing engineers to harness the full power of working together.

Unfortunately, this idea failed to realise its full potential. Researchers such as Jonathan Grudin realised that elaborate Cscw applications are just too hard to implement effectively, and we should restrict ourselves to only the most simple applications [12].

From the advent of Cscw came groupware toolkits such as GroupKit [19]. Numerous collaborative tools for general, non-structured tasks have been successfully created using such toolkits; Greg Phillips presents a comprehensive listing in [18]. For relatively focused software engineering tasks, groupware toolkits may be successfully applied to develop collaborative tools; an example of such a tool is GroupCRC, a collaborative Class-Responsibility-Collaborator (Crc) editor [5].

A handful of software engineering tools have also written using other collaborative technologies. Tukan is a collaborative SmallTalk editor and class browser [20] based on the Orwell shared source code repository [22]. Another example is Rosetta [10], a web-based tool for Java program design.

Whilst effective for their intended purposes, all of the above tools do have a significant shortcoming: they are designed for a specific task and/or a specific language. Furthermore, Cscw technology is not well suited to general software engineering; the constraints of highly syntactical structures and complex semantic relationships make persistent software artifacts difficult to develop and maintain collaboratively. Furthermore, the risk of compromising system integrity by insufficiently coordinated user actions is unacceptably high.

Additionally, most CSCW frameworks do not support transient participants, where individuals can enter and leave meetings as they please, and/or establish new meetings as needed. CSCW frameworks normally stipulate strict floor control policies; often only one person can make a modification and/or communicate at a time.

The characteristics of CSCW technologies listed above make it very difficult to create CSCW based software engineering tools. Not surprisingly, today's most popular design and development tools such as Microsoft's Visual Studio and Borland's Together [9] only support collaboration through asynchronous version control systems. Accordingly, asynchronous development by way of version control systems is by far the main means of collaborative development, as asserted by Ye and Kishida [25].

Version control system have been very successful over the last decade; SourceForge, for example, boasts 300,000 active programmers for 30,000 projects. The success of the Linux operating system can also be partly attributed to effective version control systems. As stated previously, however, shortcomings of version control based programming include having to lock each part of the system in order to maintain system integrity, a significant lag between modification and problem identification, and programming in relative isolation.

In an attempt to address some of the problems of version control systems, a new method of real-time software engineering collaboration is gaining widespread acceptance, namely XP (Extreme Programming). The XP methodology includes a significant pair programming component, where two users share the same computer, keyboard, and display. One of the main merits of this methodology is to extract the benefits of spontaneous synchronous collaboration, as described by Kent Beck[1]. Unfortunately, the XP methodology also has several shortcomings; the most obvious being the restriction of collaboration to within pairs of programmers.

As argued by Nardi and Millar, programming is a naturally collaborative activity, and many benefits arise from collaborative programming [17]. Whilst there are collaborative architectures for general work, such as CVW [21], none can handle the complexities of software engineering. Subsequently, to further advance the state of software engineering, real-time systems that promote spontaneous collaboration between any team of developers must be established, allowing the full advantages of group work to be leveraged.

To address this problem, we present a framework to support synchronous collaborative software engineering. The framework, CAISE (Collaborative Architecture for Iterative Software Engineering), is designed to enable the development of real-time collaborative software engineering tools. The framework supports real-time shared development of software artifacts, and is extensible to any programming language with its associated tools and notations. We believe that the CAISE framework will progress software development tools towards being collaborative in real-time, which is critical to the advance of software engineering.

The remainder of this paper is structured as follows. In the next section we outline the design principles of the CAISE framework. Section 3 provides an overview of how the framework is used, using a source code editor and class diagramming tool as a working example. Implementation details of the CAISE server are discussed in Section 4, followed by a in-depth discussion of the implementation of the example tools in Section 5. Our conclusions and future work appear in Section 6.

2 Design Principles of the CAISE Framework

Realistically sized software projects involve teams of people using several tools each to edit numerous related artifacts concurrently. The purpose of the CAISE framework is twofold. Firstly, it allows individual software engineers to work with a minimum of intervention on any part of the system with any tools. Secondly, it ensures that all cumulative updates are integrated, and users are kept aware of relevant changes to the system state.

A key characteristic of the CAISE framework is that the participating tools are not required to exclusively lock an entire file in order to edit it. Instead, each tool is sent updates of changes to all open artifacts as soon as they happen, meaning that users can immediately see any changes by others that are in their current view. This increased awareness of each others' actions and intentions enables users to coordinate their activities in a more informed manner, minimising possible conflicts.

Similarly, tools within the same project are made aware of any relevant relationships that exist between artifacts. For example, if one tool is editing a particular class, and another tool is editing a source file that makes a reference to that class, both tools will be notified that a dependency exists. Section 5.2 shows a detailed illustrated example of this feature.

Users of CAISE based tools are also immediately informed of the system-wide affects of their changes to artifacts, such as notification when a dependency between two source files is broken. It has been shown by Wilcox et. al. that immediate feedback can assist programming [24], and we believe that immediate impact reports are useful to all programmers—rather than waiting for the nightly build to determine the full consequences of a system modification.

To support the aspects of immediate feedback and synchronous editing, the CAISE framework introduces the following features:

A central model: For each software project, a model of the software is built up every time a source file is submitted. The model is internal to the CAISE server—no tool will ever access the model directly. The server uses the model to calculate the impact of impending user changes, and to detect relationships between artifacts.

A grammar and parse tree format: For every programming language supported by CAISE, a grammar is shared between the server and associated language-specific tools. The framework stores each artifact as a parse tree conforming to the relevant grammar; such trees are sent to each client when requested, enabling tools to update their own views of artifacts. The central model for each project is built up by inspecting every artifact's corresponding parse tree.

Propagating events: Every time the model is changed in a project, the CAISE server sends an updated artifact out to each tool currently viewing that artifact. The artifact contains both a parse tree and the latest source code file, along with other information such as current viewers and modification dates. Other events propagated to clients include user events such as users changing location in a file, or a user changing the integrity of the model, such as adding a previously unresolved symbol into the software project.

Extensibility of tools and supported languages: Any number of programming languages can be supported within the framework, and any tools and technologies may connect to the project as long as they conform to the project grammar.

Our experiences with CAISE have convinced us that it is a genuinely useful framework for supporting collaborative software engineering. To our knowledge, the CAISE approach is original: its combination of the ability to concurrently edit shared artifacts of any type, the provision of immediate context-based feedback, and the ability to incorporate new programming languages into a collaborative framework represents a significant step forward within the field of collaborative software engineering.

3 Using the CAISE Framework

Typical usage of the CAISE framework is presented in Figure 1, which illustrates three participating applications, known as CAISE *tools*, editing numerous shared artifacts of project 'A'. The CAISE framework allows software projects to be developed by any number of users concurrently, where each project may have any number of artifacts in the form of source files and other resources.

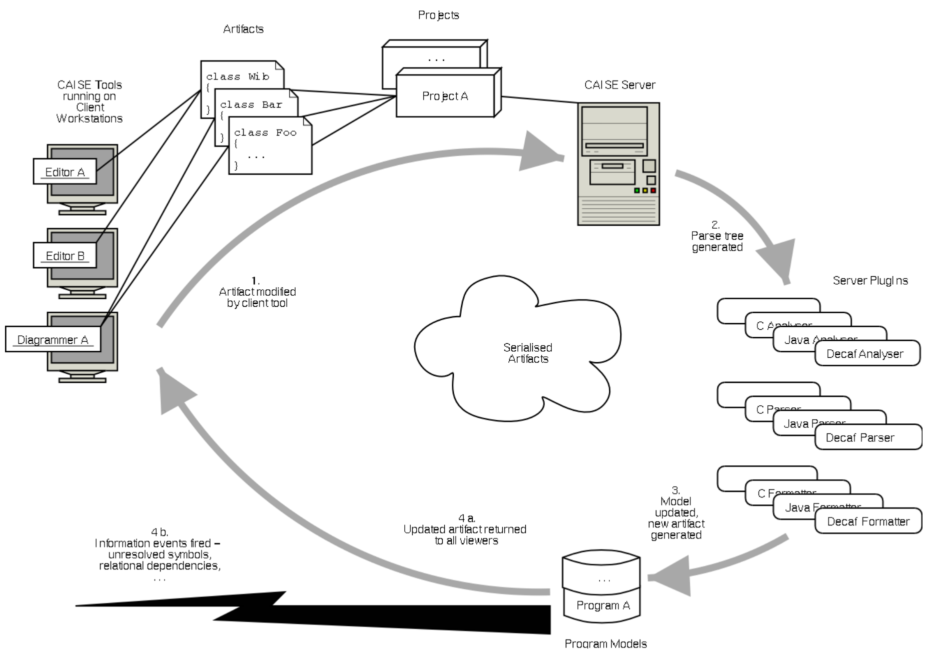


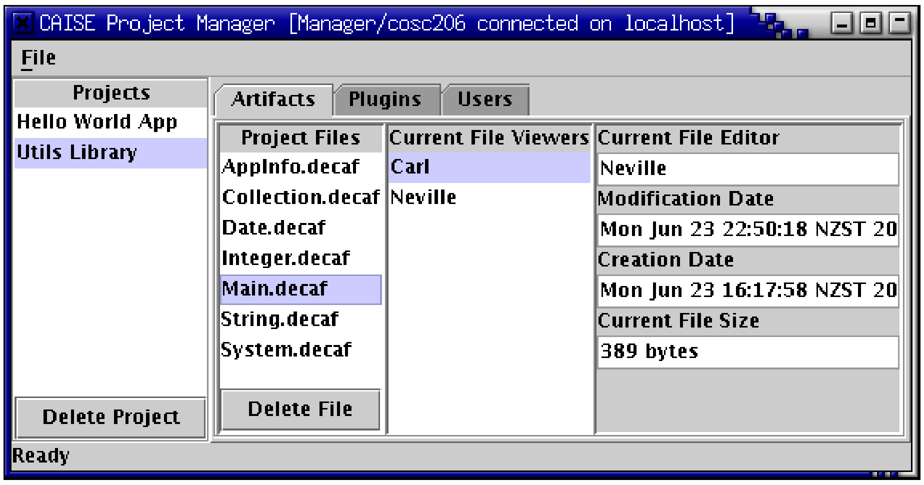
Figure 1: A high-level view of the CAISE architecture

To accommodate multiple programming languages, the server loads language-specific components during system startup, known as CAISE *plug-ins*. Plug-ins, such as those illustrated in Figure 1, are typically in the form of parsers, parse tree semantic analysers, and source code formatters (see Section 5). Parsers turn source files into parse trees, semantical analysers build an in-memory model of software based from the project parse trees, and source formatters reverse-engineer source files from the project model.

3.1 Tour of a Caise Project

Here we present a brief tour of the framework, showing how to create a Caise based project and commence collaborative work. For this example, we are not concerned with implementing Caise plug-ins to support a new language—we will assume that such components already exist. For details on how to extend the framework with a new language, please refer to Section 5.4.

To configure a new or existing project, we use the Project Manager component. Figure 2 demonstrates the Project Manager being used to inspect the project ‘Utils Library’. To create a new project, the administrator simply selects the appropriate item from the Project Manager menu, and supplies a unique project name.


CAISE Project Manager [Manager/cosc206 connected on localhost]
File
Artifacts
Plugins
Users
Delete Project
Delete File
Ready

Projects
Hello World App
Utils Library

Project Files	Current File Viewers
AppInfo.decaf	Carl
Collection.decaf	Neville
Date.decaf	
Integer.decaf	
Main.decaf	
String.decaf	
System.decaf	

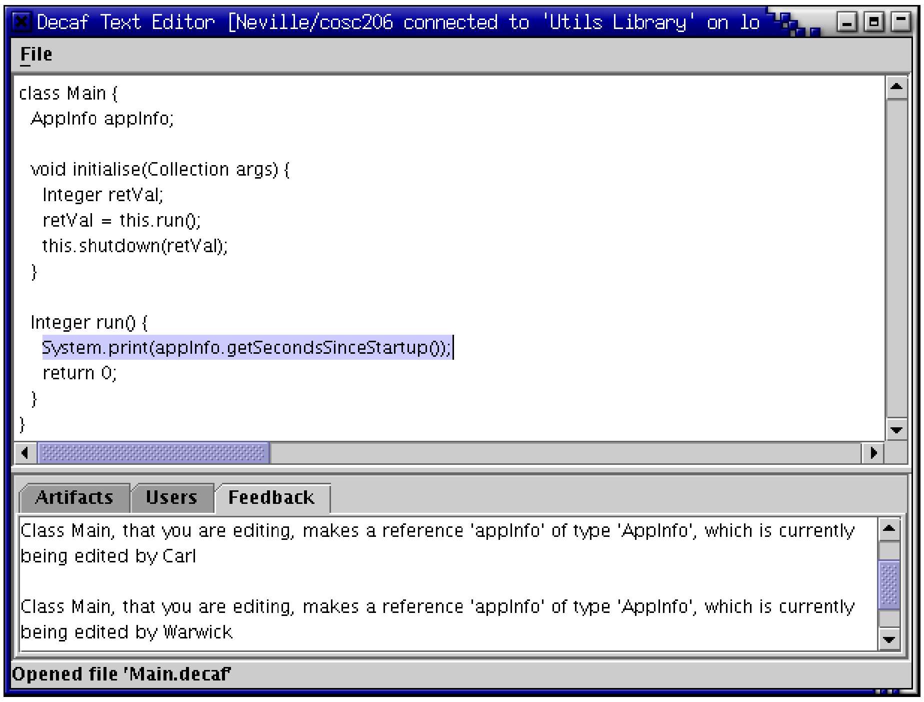
Current File Editor: Neville
Modification Date: Mon Jun 23 22:50:18 NZST 20
Creation Date: Mon Jun 23 16:17:58 NZST 20
Current File Size: 389 bytes

Figure 2: The Caise Project Manager component being used to inspect the ‘Utils Library’ project

Each project is configured to use an analyser; typically this is done at project creation time through the ‘PlugIns’ panel of the Project Manager component. For each artifact, the analyser determines which parser and source formatter plug-ins to use. The project in the current example has been configured to use an existing analyser for the Decaf language; further details of this language and associated plug-ins are given in Section 5.1.

To commence collaborative work on a project, we must use tools that conform to the Caise framework. For the purpose of this illustrated section, our examples use existing Caise tools written for the Decaf language (see Section 5.2). For information on the creation of new tools for the Caise framework, please refer to Section 5.3. These tools are presented for illustratory purposes, and whilst they allow us to demonstrate the Caise framework, they do not represent the complete capabilities of Caise based tools.

Figures 3, 4, and 5 present three running instances of Caise tools connected to the 'Utils Library' project. As can be seen from the Project Manager component in Figure 2, this project contains several artifacts, with some artifacts being currently modified. Neville has opened the artifact 'Main.decaf' with a code editor (Figure 3), Carl is using the same type of editor as Neville to work on the artifact 'AppInfo.decaf' (Figure 4), and Warwick is editing the same artifact as Neville , but he is using a class diagramming tool to do so (Figure 5).


Decaf Text Editor [Neville/cosc206 connected to 'Utils Library' on lo
File
Artifacts
Users
Feedback

```
class Main {
  AppInfo appInfo;

  void initialise(Collection args) {
    Integer retVal;
    retVal = this.run();
    this.shutdown(retVal);
  }

  Integer run() {
    System.print(appInfo.getSecondsSinceStartup());
    return 0;
  }
}
```

Class Main, that you are editing, makes a reference 'appInfo' of type 'AppInfo', which is currently being edited by Carl

Class Main, that you are editing, makes a reference 'appInfo' of type 'AppInfo', which is currently being edited by Warwick

Opened file 'Main.decaf'

Figure 3: The Decaf source code editor being used by Neville to edit 'Main.decaf'

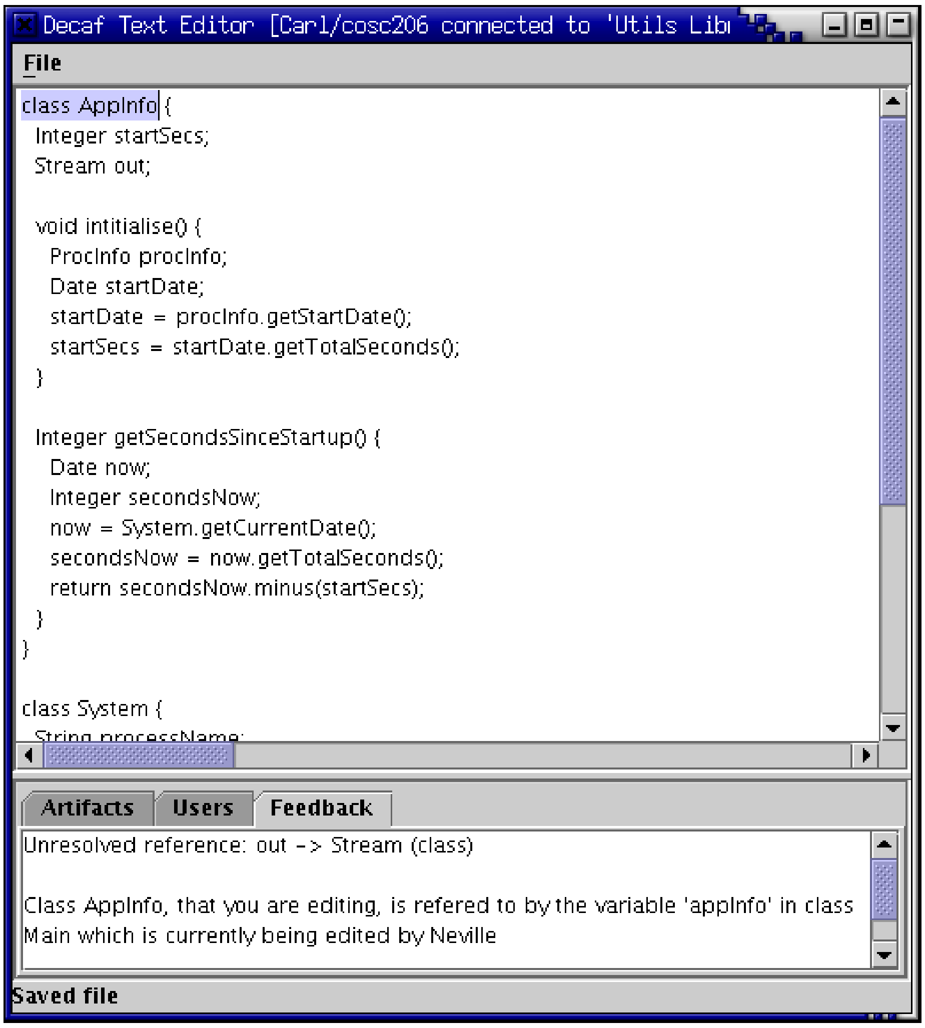
Whenever an artifact is opened or modified, each user receives feedback via the control panel component, as shown in the related figures. This feedback can be displayed in any arbitrary manner. Each time an artifact is modified, the model is updated, and the corresponding Caise tools update their displays and present any associated feedback. The example presented in this section illustrates such feedback, reinforcing the key design principles introduced in Section 2:

Artifacts can be viewed and edited concurrently even with multiple views.

Unresolved references are displayed as soon as they are encountered.

Dependencies are highlighted relevant to open artifacts.

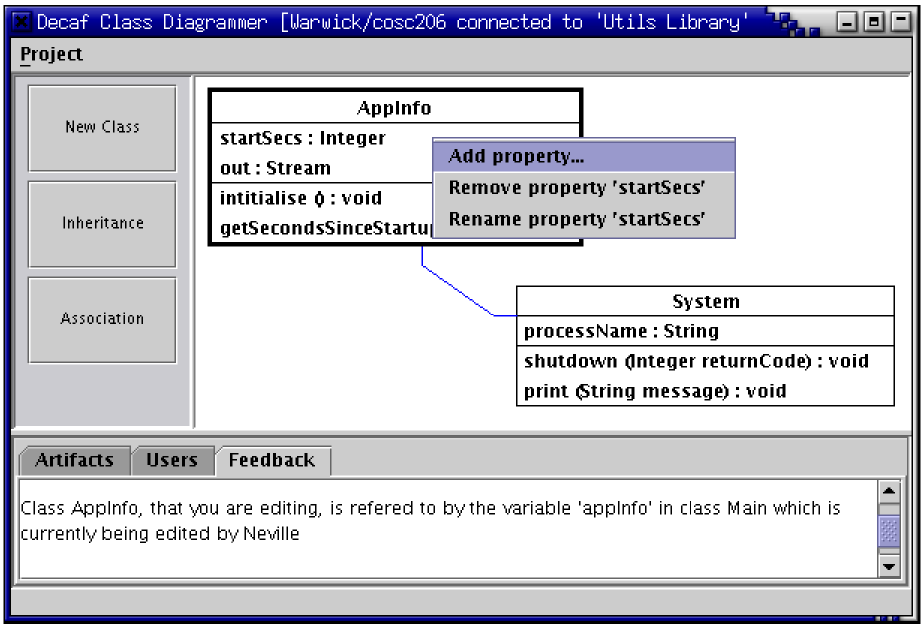
This example also illustrates communication between users within a Caise project. To assist communication, the control panel component also provides a



```
class AppInfo {
  Integer startSecs;
  Stream out;

  void intitialise() {
    ProcInfo procInfo;
    Date startDate;
    startDate = procInfo.getStartDate();
    startSecs = startDate.getTotalSeconds();
  }

  Integer getSecondsSinceStartup() {
    Date now;
    Integer secondsNow;
    now = System.getCurrentDate();
    secondsNow = now.getTotalSeconds();
    return secondsNow.minus(startSecs);
  }
}

class System {
```

```
Unresolved reference: out -> Stream (class)

Class AppInfo, that you are editing, is refered to by the variable 'appInfo' in class
Main which is currently being edited by Neville
```

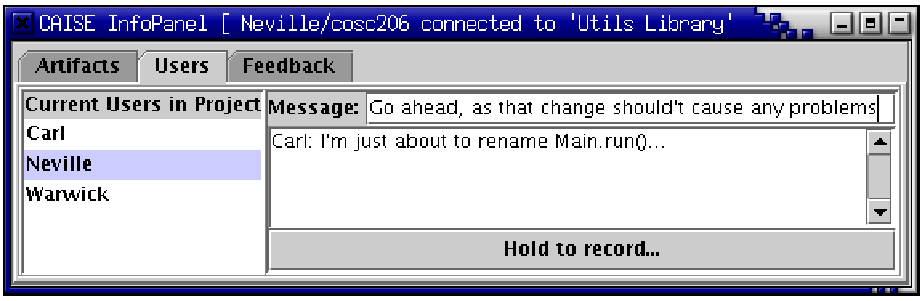
Figure 4: The Decaf source code editor being used by Carl to edit 'AppInfo.decaf'


Decaf Class Diagrammer [Warwick/cosc206 connected to 'Utils Library'
Project
New Class
Inheritance
Association
AppInfo
startSecs : Integer
out : Stream
intitialise () : void
getSecondsSinceStartu
Add property...
Remove property 'startSecs'
Rename property 'startSecs'
System
processName : String
shutdown (Integer returnCode) : void
print (String message) : void
Artifacts
Users
Feedback

Class AppInfo, that you are editing, is refered to by the variable 'appInfo' in class Main which is currently being edited by Neville

Figure 5: The Decaf class diagrammer being used by Warwick to edit 'AppInfo.decaf'

'Users' panel, which enables text and audio messaging. This panel can be seen in Figure 6. When a CAISE tool connects to a project, the associated user is subscribed to the internal project channel, allowing him or her to send both text and voice messages to other selected users or the entire group in an unobtrusive manner.



Go ahead, as that change should't cause any problems

Carl: I'm just about to rename Main.run()...

Figure 6: The Control Panel component of the CAISE framework being used by Neville to communicate with Carl

4 Inside the CAISE Server

Within the CAISE framework, the server is a key component, enabling real-time collaborative development of the artifacts it maintains for each project. In this section, we discuss the mechanisms of the CAISE server, including how it builds up an internal software model, how it facilitates the concurrent editing of artifacts, and how it communicates with remote tools.

The CAISE server holds several types of information for each project. Each artifact is stored on the server, including additional details such as the clients currently editing it, and the artifact's modification and creation dates. The software model for each project is also held and maintained on the CAISE server. As mentioned in Section 2, this model authoritatively defines the software being developed within the project.

In further detail, the model holds all the source code declarations made by each project's artifacts, in enough detail to build the project as machine instructions, or reverse engineer the project back into a collection of artifacts. In terms of compiler technology, the model is effectively a type-checked and semantically analysed collection of parse trees for the entire project. The model represents every component of the program being developed, including classes, properties, methods, statements, and references. For the purposes of the CAISE framework, the model is used to determine dependencies between open artifacts.

Given that the server-maintained model represents the core of each software project, the CAISE framework implements the *Model-View-Controller* design pattern [8]. The CAISE server represents the controller—allowing modifications to the model, and each CAISE tool maintains its own view of the model.

The next aspect of the server to describe is the mechanism for distributing the model to each CAISE tool. The most obvious approach is to serialize the model in its entirety to send to each tool, but it turns out that this is not necessary. In the CAISE framework, the server actually sends artifacts out to each CAISE tool; each tool can retrieve the source code buffer and the parse tree from the artifact for display purposes, along with other information such as current viewers and modification details.

For many tools, an updated source code buffer is sufficient - every time an artifact is modified, the updated version is received immediately. For other tools, such as class diagrammers, a parse tree is required. Parse trees are trivial to inspect when looking for specific entities such as class and member declarations; this is discussed further in Section 5.2.

We now discuss how the project model is updated within the CAISE framework. As illustrated in Figure 7, each CAISE tool maintains its own local version of the server model, in a manner representative of the *Proxy* design pattern [23]. For the CAISE framework, initially each tool has its own copy of every artifact. Upon editing of an artifact, the artifact is immediately sent to the CAISE server; this can be seen the second time-sequence of Figure 7.

Continuing with the time-sequence of Figure 7, the server then invokes a parser over the modified artifact to generate an updated parse tree. The analyser then integrates this parse tree into the model, updating the model accordingly. Finally, the updated artifact is redistributed to every relevant CAISE tool in the project, allowing each local model (if any exist) to be updated, and each associated artifact to be redisplayed.

4.1 The CAISE Communication Layer

Until now, no mention has been made as to how the CAISE server and tools communicate with each other. To facilitate communication, one sub-layer of the

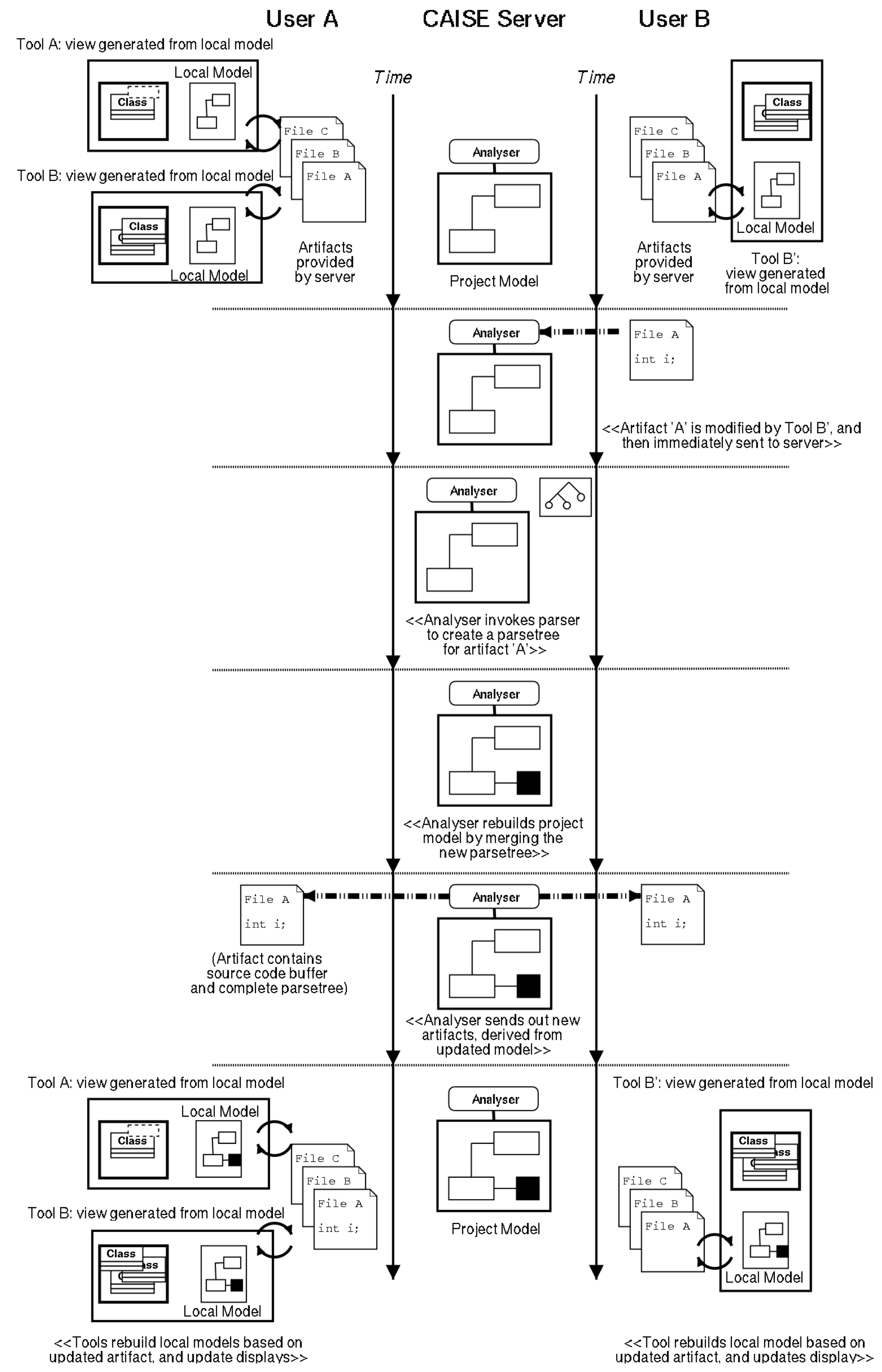

User A
CAISE Server
User B
Time
Time
Tool A: view generated from local model
Local Model
Class
File C
File B
File A
Tool B: view generated from local model
Local Model
Artifacts provided by server
Analyser
Project Model
Artifacts provided by server
Tool B': view generated from local model
Local Model
Analyser
File A
int i;
<<Artifact 'A' is modified by Tool B', and then immediately sent to server>>
Analyser
<<Analyser invokes parser to create a parsetree for artifact 'A'>>
Analyser
<<Analyser rebuilds project model by merging the new parsetree>>
File A
int i;
Analyser
File A
int i;
(Artifact contains source code buffer and complete parsetree)
<<Analyser sends out new artifacts, derived from updated model>>
Tool A: view generated from local model
Local Model
Analyser
Tool B': view generated from local model
Tool B: view generated from local model
Project Model
Local Model
Local Model
<<Tools rebuild local models based on updated artifact, and update displays>>
<<Tool rebuilds local model based on updated artifact, and updates display>>

Figure 7: Schematic view of an artifact modification within the CAISE framework

CAISE architecture provides a messaging framework for groups of applications.

The messaging framework is in the form of a Pure-Java API, and allows groups of applications to communicate via synchronous method calls. Through the use of the messaging framework, locality of the tools and the CAISE server is transparent—all processes could run on the same machine, different machines within a local network, or spread across the Internet.

The messaging framework is based loosely on the Java Shared Data Toolkit [3], and provides several multi-user facilities such as the audio 'TalkButton' component presented in Section 3.1. Full design and implementation details of the messaging framework are presented in Cook et. al. [6].

5 Inside CAISE Tools and Plug-ins

In this section we take a closer look at the CAISE tools presented in Section 3. We explain how the tools are implemented, how to implement new tools, and how to introduce a new programming language into the CAISE framework.

5.1 Custom Grammars and Components

The tools presented in Section 3 conform to a CAISE supported grammar called Decaf. Decaf is a simple subset of the Java language, created to assist us in illustrating the CAISE framework. Decaf is powerful enough to write useful programs, but not so complicated that it becomes impossible to explain the details of CAISE compliant tools.

To describe the grammar, each source file may have any number of classes just like Java, although there are no inner or nested classes, and there are also no interfaces. Each Decaf class can have any number of methods and properties, and methods may declare variables, call other methods, make assignments, and declare return statements. Figure 4 shows a typical Decaf source file.

For the CAISE framework to support the Decaf language, corresponding server plug-ins were required. A parser plug-in that conformed to the Decaf grammar was the first component developed. This was written easily using the CUP parser generator [13], and the JFLEX lexical scanner [15]; we provided a grammar for Decaf and the rules for building up a parse tree, and CUP generated a Decaf-compliant parser.

A source formatter also had to be implemented in order for the CAISE framework to support the Decaf language. A source formatter is essentially the opposite of a parser—it reconstructs a source file from the parse tree. Source formatters are typically used within the CAISE framework to supply a source file to a source code editor, based off an artifact initially generated by a class diagramming tool. Again, this component was trivial to implement; the formatter simply walks through the given Decaf parse tree and writes out code statements based on the elements it finds. We found the Visitor design pattern particularly suitable for this task [8].

The final component necessary to support Decaf within the CAISE framework is that of the project semantic analyser. Unlike parser and source formatter components, analysers require substantially more effort to implement, regardless of the language being supported. In the case of Decaf, the semantic analyser was written in approximately 200 lines of code. The main functions of the analyser were to build up a model of the project software based from the artifact parse trees, type-check and semantically analyse the model, and detect any dependencies by inspecting the model, given the artifact locations of every active user.

5.2 Mechanisms of Caise Tools

In this section, we discuss the implementation details of the Caise tools presented in Figures 3, 4, and 5.

For both the Decaf source code editor and Decaf class diagramming tool, we started development specifically as single user applications, then integrated the services of the Caise Api as the final step. The Api is used by both applications in order to connect to the Caise server, open existing projects, create new artifacts, and open existing artifacts in order to retrieve the latest source file buffers and parse trees.

Typically, artifacts are saved via the Api upon every significant change—for example a new line of code in a source code buffer, or any new event in a class diagram. If the server runs on fast hardware, character-by-character saves are possible without any noticeable delay; we asserted this when using a dual Pentium-4 server with 1 gigabyte of primary memory. To avoid any transactional errors when modifying source files, each source code editor may send an update of the buffer on a character by character basis. This allows every other viewer of the artifact to immediately see any changes pending, and possibly discuss the change between developers before the update is committed.

Upon a Caise tool saving a source file, the update will immediately be delivered to all current viewers of the artifact, as described in Section 4. Each tool will then extract either the source code buffer or the parse tree from the artifact, and update the local model. For the Decaf source code editor, this simply involved redisplaying the source file. For the Decaf class diagramming tool, this meant walking through the parse tree, and redisplaying the classes, methods, properties, and references as they were discovered. As this task involves iterating through the elements of the parse tree in a similar to that of the Decaf source formatter component, much of the code is common between the programs.

When two or more instances of a Caise tool work on the same project, the Caise server is likely to detect dependencies between currently opened artifacts. Similarly, whenever the state of the program changes, for example a reference is made to a class that does not yet exist, the Caise server will also take notice. Whenever such information is gathered, the server will send a message out to every Caise tool that requires notification.

We can see an example of this in Figures 3 and 4. In Figure 3, user Neville is editing the file 'Main.decaf' that declares a property `appInfo` of type `AppInfo`. Concurrently, in Figure 4, user Carl is editing the class 'AppInfo.decaf' that declares and defines the type `AppInfo`. The server generates an appropriate warning to both users, which is delivered to each Caise tool for application-defined processing.

For users where collaborative feedback is not required, applications may choose to ignore feedback messages from the server. A balance between enforcing system integrity through conflict management and delegating responsibility through social protocols (as described by Uwe Busbach [4]) is maintained at the server. Consequently, the entire system may be effectively locked for a period during conflict resolution. In the case of the Decaf tools, the messages are simply displayed in the 'Feedback' region of the Control Panel component, as presented in Section 3.

5.3 Extensibility: Creating New Caise Tools

Creating new tools for a Caise supported programming language is a relatively simple process. As described the previous section, the easiest way to create a new Caise tool is to start with a single user tool, and then integrate the collaborative services of the Caise Api. The Api may be used to connect to the Caise server, and obtain lists of available projects and artifacts. Caise compliant software engineering tools simply need to notify the server every time an artifact is modified or saved, and redisplay artifacts every time a change is made remotely.

The Caise Api is intended to minimise the programming effort required to develop a new collaborative tool, allowing tools to communicate with the Caise server, and other tools. Additionally, several other Caise components are provided to assist the rapid development of collaborative tools. The Control Panel allows navigation of artifacts and users for the current project, and also facilities text and audio communication between project members. Multi-user widgets, such as the TalkButton, allow the development of custom interfaces for more elaborate applications.

The Caise Api is written in Java. This means that only Java-based applications can be developed for use with the Caise framework. It also means that all server plug-ins must be written in Java or Java-supported languages such as C++. We do not consider this to be a negative characteristic of the Caise framework—Java has gained widespread acceptance throughout the research community as a platform-independent language, and a commercial version of the Caise system could easily support multiple languages and technologies such as Sun Microsystems' J2EE architecture and Microsoft's .net framework.

5.4 Extensibility: Supporting a New Language

As explained in Section 5.1, for a language to be supported by the Caise framework, the core server components for that language must be implemented. The degree of effort required to support a new language is proportional to the complexity of the language. Whilst scanners and parsers are relatively trivial to create for any carefully planned language, semantic analysers are complex and time consuming to develop. For the Decaf language, the analyser component only required a day to develop, but for a more complex language, considerable effort will be required.

To demonstrate that supporting a more complex language is possible, meaningful, and useful, we are currently extending the Caise framework to accommodate the Java language. To do so, we require a full Java semantic analyser; in other words, the complete compiler front-end for Java, including the type checker and ambiguous symbol resolver. We intend to use the Java analyser as described by Irwin et. al. [14]. By making minor modifications to this analyser, it is possible to wrap it as a server plug-in, and implement Caise tools to support the Java language.

6 Conclusions & Further Work

In this paper, we have presented an architecture for supporting real-time collaborative software engineering by developers located anywhere on the Internet. The key features of the approach are:

Grammar-based tools and models: User-level tools, such as diagrammers and editors, work with appropriate level views of local models of the corresponding artifacts. Both the models and views are based on grammars. For example, a simple editor might represent a Decaf class in terms of paragraphs, lines and words while a more powerful editor might represent the same class in terms of declarations, expressions and other terms from the full Decaf grammar.

Change integration and analysis: Changes generated by individual tool users are sent to the Caise server which maintains current parse trees for all project artifacts. The Caise server integrates changes, and resolves (potential) conflicts.

Upgrade propagation and awareness: Semantic analysers react to changes in relevant portions of the project parse trees and initiate the propagation updates to the local models of individual users.

Flexibility and extensibility: The use of plug-ins enables customisation of the behaviour of analysers to support arbitrary awareness notification mechanisms or to respond to new patterns of updates. The addition of new tools, to allow for new languages or notations, simply requires the development of an appropriate grammar, parser, and analyser along with the tool interface. The Caise Api provides support for developers of such tools.

A selection of prototype tools for the Decaf language has been described to illustrate the major aspects of the implementation of the architecture. These provide:

- awareness of other users and their activity, together with text and audio facilities to augment individual tools with real-time group discussion.
- support for editing, including feedback on the impact of changes to project artifacts resulting from the activities of other users, regardless of the particular tools used.
- support for diagram-based development to complement text editing. Individual users may update artifacts using either, or both, editor or diagrammer tools: updates are integrated and propagated back to the users irrespective of the tools used to initiate them.

The Caise architecture could be used effectively for supporting flexible single-user development tool sets. However, we are primarily interested in supporting communities of developers collaborating in real time.

Having demonstrated the feasibility of implementing the architecture, we are currently developing tools for Java and Uml in order to illustrate the extension to "real" programming languages and techniques. This will also allow our approach to be compared with alternative tools.

The major thrust of our work to date has been the towards the development and implementation of the architecture, and we are encouraged by its performance so far. However, it is also important to evaluate its potential to be of real use to physically separated groups of developers. Evaluation has several aspects including:

- scalability of the CAISE server.
- ability to maintain project integrity.
- usability of individual tools and sets of tools.
- provision of sufficient awareness of the locations, intentions and actions of others that real-time collaboration is effective.
- extensibility.

Currently, only anecdotal feedback has been obtained. However, we plan to conduct more formal trials as development proceeds to the stage where the tool functions match our users' experience more closely.

Our experiences with CAISE have convinced us that it is a genuinely useful framework for supporting collaborative software engineering.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, Reading, MA, 1 edition, October 1999.

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

[3] R. Burridge. *Java Shared Data Toolkit User Guide.* Sun Microsystems, October 1999. Online document, Available from *java.sun.com/products.*

[4] U. Busbach. *Activity Coordination in Decentralized Working Environments*, chapter 8. In Dix and Beale [7], September 1996.

[5] N. Churcher and C. Cerecke. GroupCRC: Exploring CSCW Support for Software Engineering. In *Proceedings of the 4th Australasian Conference on Computer-Human Interaction*, Hamilton, New Zealand, November 1996. IEEE Computer Society Press.

[6] C. Cook and N. Churcher. A Pure-Java Group Communication Framework. Technical Report TR-COSC 02/03, Department of Canterbury, University of Canterbury, Christchurch, New Zealand, June 2003.

[7] A. Dix and R. Beale, editors. *Remote Cooperation: CSCW issues for Mobile and Tele-workers.* Springer/BCS, September 1996.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[9] C. Garrett. Software Modeling Introduction: What Do You Need from a Modeling Tool? Borland Software Corporation White Paper, 28 May 2003.

[10] N. Graham, H. Stewart, A. Ryman, R. Kopaee, and R. Rasouli. A World-Wide-Web Architecture for Collaborative Software Design. In *Software Technology and Engineering Practice*, Pittsburgh, Pennsylvania, August 30 - September 02 1999. IEEE.

[11] S. Greenberg. The 1988 Conference on Computer-Supported Cooperative Work: Trip Report. In *SIGCHI Bulletin*, volume 20 of *5*, pages 49–55. ACM, July 1989.

[12] J. Grudin. *Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces*, pages 552–560. In Marca and Bock [16], 1992.

[13] S. E. Hudson. *LALR Parser Generator for Java.* Visualization and Usability Center, Georgia Institute of Technology, Atlanta, GA, July 1999.

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

[15] G. Klein. *JFlex Version 1.4, The Fast Lexical Analyzer Generator for Java.* Munich, Germany, March 2003. Available from `www.jflex.de`

[16] D. Marca and G. Bock, editors. *Groupware: Software for Computer-Supported Cooperative Work.* IEEE Press, Los Alamitos, CA, 1992.

[17] B. A. Nardi and J. R. Miller. An Ethnographic Study of Distributed Problem Solving in Spreadsheet Development. In *Proceedings of the Conference on Computer Supported Cooperative Work*, pages 197 – 208, Los Angeles, CA, October 7-10 1990. ACM.

[18] W. G. Phillips. Architectures for Synchronous Groupware. Technical report, Department of Computing and Information Science, Queen's University, Ontario, Canada, May 1999.

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

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

[21] P. Spellman, J. Mosier, L. Deus, and J. Carlson. Collaborative Virtual Workspace. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 197–203, Phoenix, AZ, November 1997. ACM Press, NY.

[22] D. Thomas and K. Johnson. Orwell: A Configuration Management System for Team Programming. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (SIGPLAN)*, pages 135–141, San Diego, CA, 1988. ACM.

[23] J. Vlissides. *Pattern Hatching: Design Patterns Applied.* Addison-Wesley, Reading, Massachusetts, 1998.

[24] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems? In *Proceedings of CHI 97: Human Factors in Computing Systems*, Atlanta, GA, 22-27 March 1997. ACM.

[25] Y. Ye and K. Kishida. Toward an Understanding of the Motivation of Open Source Software Developers. In *Proceedings of the International Conference on Software Engineering (ICSE2003)*, Portland, Oregon, May 3-10 2003.