A User Evaluation for Synchronous Collaborative Software Engineering Tools

Carl Cook Warwick Irwin Neville Churcher

Technial Report TR-04/05, July 2005
Software Engineering & Visualisation Group,
Department of Computer Science and Software Engineering,
University of Canterbury, Private Bag 4800,
Christchurch, New Zealand
{carl, wal, neville}@cosc.canterbury.ac.nz

Abstract

Collaborative Software Engineering (CSE) is a rapidly growing field of research, with commercial tools starting to incorporate new collaborative features into their currently single-user products. We have undertaken an empirical evaluation to investigate the envisaged benefits of code-level collaboration for tools that we have developed. Results of our evaluation show that for two typical programming scenarios, collaborative tools achieve task completion rates that are at least twice as fast as their conventional counterparts. Additionally, for a number of subjective aspects, the participants were strongly in favour of using the new tools. From these results, we are encouraged to continue developing our collaborative tools, and to investigate other aspects of such tools within the context of CSE.

Contents

1 Introduction **3**

2 Background **4**

3 Tool Overview **5**
3.1 Controlling Crosstalk During Compilation 6
3.2 Implementation Notes 8

4 Evaluation Method **8**
4.1 Aim and Purpose 9
4.2 Participants 9
4.3 Physical Layout 10
4.4 Apparatus 10
4.5 Procedure 11
4.5.1 Experimental Design 11
4.5.2 Supporting a Minimal Code Repository Interface 13
4.5.3 Tool Modes 14
4.5.4 Task Types 14
4.5.5 Order of Groups and Tasks 15
4.5.6 Training Manual 16
4.5.7 Evaluation Tasks 17
4.5.8 User Survey 17
4.5.9 Statistical Validity 17
4.6 Summary 20

5 Evaluation Results **20**
5.1 Task Completion Times 21
5.2 Subjective Assessment 21
5.3 User Preferences 22
5.4 User Comments 22

6 Threats to Validity **23**

7 Discussion **24**

8 Conclusions and Future Work **25**

A Evaluation Documents **28**
A.1 Training Manual 28
A.2 Training Tasks 29
A.3 Evaluation Tasks 30
A.4 Participant Surveys 31

B Source Code **31**

1 Introduction

Software Engineering appears to be more demand-driven from industry than most other areas of Computer Science research. This can be seen by the adaptation of object-oriented programming—a paradigm changing event that was initiated on perceived benefits and positive case studies rather than by prior scientific research. Very little empirical research has been conducted related to code-level collaboration, yet both Integrated Development Environment (IDE) designers and end users appear enthusiastic about the current trend of support for collaborative development.

It is therefore a worthy task to undertake a user evaluation into Collaborative Software Engineering (CSE) in order to investigate and solidify the perceived benefits from the aspect of empirical software engineering research. We are also well suited to perform an evaluation at this point in time, as we have just completed the development of a new set of CSE tools which were originally reported in [5, 8].

The main premise of our research is that enabling more collaborative software engineering through advanced tool support will in turn raise the very restricted levels of communication within current software engineering practice. To validate this premise, we will compare CSE tools with their conventional counterparts, with the aim of showing scenarios where the collaborative tools are better.

The term *better*, however, is difficult to define objectively within empirical software engineering research. The definition of better for CSE tools can have many meanings—faster task completion rates, easier to use, fewer bugs in the resultant programs, more tolerance to large groups, encouragement of greater communication between programmers, better program comprehension, better awareness of other programmers' changes to name a few. Additionally, it is difficult to define the range of allowable values for external factors that affect the evaluation, such as size, type and difficulty of evaluation tasks, experience of participants, tools to be used within the control group, and features of the tools being evaluated.

Accordingly, it appears highly challenging to design a test that can evaluate all aspects of software engineering within a single context. We have therefore limited the evaluation presented in this paper to the objective measurement of task completion rates for mechanically scripted tasks between pairs of collaborating users, as well as gathering subjective measures such as user preferences. Our hypothesis is therefore that collaborative tools give task completion rates superior to those of their conventional counterparts for selected typical coding scenarios.

This paper gives full details of the experiment, including the methodology, environment configuration and results. It is intended that this paper provides enough details so that the experiment can be reproduced by other researchers for comparison against other systems and types of users.

The remainder of the paper is structured as follows. Section 2 gives a brief overview of CSE and related tools. Section 3 presents the CSE tools used within the evaluation, including design and implementation details. Section 4 describes the evaluation method in full detail, and section 5 reports the results from the user trial. Section 6 discusses the key threats to validity of the results, and section 7 provides a discussion of the entire evaluation. Finally, conclusions and

topics for future work are presented in section 8.

2 Background

In the last year many of the major commercial IDEs have taken significant steps towards code-level real time collaboration. Of the four Java IDEs that have the largest market shares, two of them now support shared development facilities, and all four environments are promising more to come in the next major releases.

Eclipse [10] is arguably the most popular development environment for Java, and has the support of many of the industry's largest corporations. While Eclipse itself does not support code-level collaboration, a new subproject called the Eclipse Communication Framework [17] aims to allow the eclipse code repository and project model to be shared and collaboratively edited. The API to perform basic sharing is available now, along with some example client applications.

Borland's JBuilder [9] is another of the main IDEs in the Java development market. It terms of collaboration it supports real-time remote refactoring and integrates the StarTeam project management suite. Similarly, Sun's JSE [16] already supports a collaborative code editor and instant messaging channels, with more plans for code-level collaboration in the next release.

Aside from fully-featured IDEs, many specialist tools support collaborative modes of work. Poseiden enterprise edition, for example, allows the authoring in real time of UML documents by any number of users in a distributed setting [2]. There are also collaborative plug-ins available for Oracle and Rational's IDEs, bringing them into the market for code-level collaborative development tools.

Within the field of research, there are numerous specific and ambitious software engineering tools to accommodate a range of tasks. For collaborative change impact reporting the Palantir architecture exists [21]. To visualise the activity of large shared code bases, the Augur visualisation suite may be used [11]. For web-based shared UML editing, Rosetta is a well known tool [12], and for distributed eXtreme Programming a new framework called Moomba has been recently been published [20].

A tool that is indirectly related to CSE yet potentially very influential is SubEthaEdit [19]. This is a shared text editor for the MacOS operating system, and it recently won the MacOS X Innovators Award for best application. It has no specific Software Engineering abilities except for basic syntax and method highlighting, but if it continues to gain popularity with software developers then IDEs might have to implement similar interfaces due to user demand.

Despite the flow of ideas for collaborative tools and frameworks, there have been very few empirical evaluations into CSE tools, technologies and concepts. While a multitude of empirical research has been published for the fields of Groupware and Human-Computer Interaction, CSE tools can be more ambitious in design and harder to evaluate. Even at a broader level, there is not a lot of empirical research into general Software Engineering when compared to related fields.

This background has listed selected tools and research projects related to CSE. For a detailed annotated bibliography on the CSE tools mentioned above, and related areas of research, please refer to [4].

3 Tool Overview

The CAISE architecture, as described originally in [5, 8], allows for the rapid development of fully featured CSE tools. Before discussing the evaluation of two CAISE-based tools as described in this paper, this section introduces the latest version of CAISE tools, their basic functionality, and the background concepts to synchronous collaboration.

The authors of the Concurrent Versioning System (CVS) say *"CVS is no substitute for communication"* [1]. We concur that other code repository systems do no better. Therefore, the basis for the CAISE set of collaborative software engineering tools was to allow programmers to work collaboratively without sacrificing communication. The CAISE tools achieve this by keeping all programmers group synchronised in real time, and at the same time providing user awareness and project state information to the individual tools.

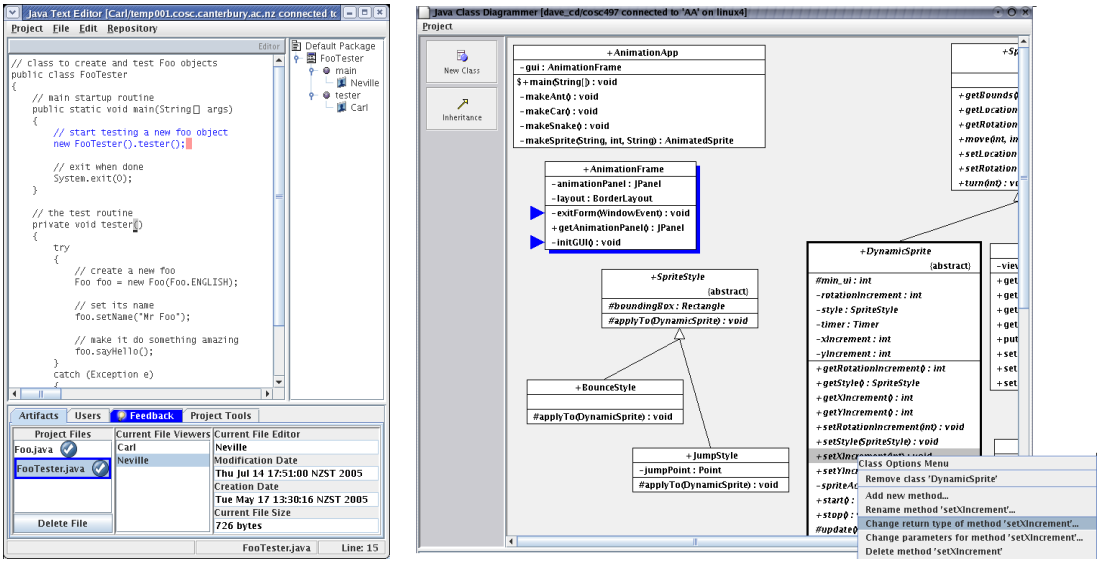
The CAISE architecture and supporting tools do not aim to replace systems such as CVS; the ability to work in private at times and to be able to keep different versions of programs separate are features that very few programmers could do without. Our tools are designed support what code repositories do not provide: communication between developers and tools during fine-grained real time collaboration.

The CAISE infrastructure does not impose a specific methodology onto CAISE-based tools, but tool developers can implement particular methodologies if and when required.

Our system is ideal for distributed pair programming. We call it *N-programming* as there is no theoretical limit to the number of people and types of tools that can collaborate at any point in time. This is a significant advance over conventional pair programming where up until now, collaborative technology limitations have restricted programmers considerably.

Over the last twelve months we have worked intensively on two existing CAISE-based tools to provide a realistic Software Engineering environment; these tools are presented in figure 1. Recent improvements include:

- A relaxed-WYSIWIS display for all tools
- Round-trip engineering between all tools
- Collaborative undo within the editor
- An adjustable level of collaborative scope when compiling the project
- The artifacts panel now displays the current compilation state of each artifact
- The editor provides remote modification highlighting and *tele-carets*
- Considerable user interface improvements have been made to accommodate the constantly changing state of each users display
- CVS has been integrated to allow a CAISE project to access a central code repository



(a) A Java code editor. When viewed in colour the remote text highlighting and tele-carets are visible.

(b) A UML class diagrammer. Remote user positions are indicated by blue markers.

Figure 1: CAISE development tools with CSCW awareness support.

By incorporating a CVS into the tools, a large set of developers can be partitioned into several CAISE sub-projects, with CVS employed to synchronise between sub-projects. See [7] for more details on this concept.

Please refer to [7] for the full documentation on the latest version of these tools, how they maintain synchronisation with the CAISE server, and how to develop new CAISE-based Software Engineering tools. Demonstrations of the tools as they execute numerous tasks are available from `www.cosc.canterbury.ac.nz/clc/cse`. This includes a demonstration of the tools operating in conventional mode as they encounter and resolve a typical merge conflict.

3.1 Controlling Crosstalk During Compilation

Unexpected real time code modifications by other users, whilst surprising, do not significantly degrade a developers ability to work within a collaborative setting. If one developer is working on the same line of code as another developer, it is likely to be beneficial if both parties pause and discuss the current activities, although programmers may choose to ignore the presence of others and carry on development. A major problem with real time development, however, is that of compiling code during a time of concurrent development activity.

Ideally, if one developer makes a change that is unrelated to the area of the program that another developer is currently working on, the second developer should not necessarily be placed in a position where he or she is prevented from compiling. This principle, known as private work, is one of the key elements to CVS and related repositories. For CSE tools, however, if the first developer has not completed their changes, or their changes are syntactically or semantically incorrect, the project will fail to complete its build even for the second user

as the entire project is shared in real time. To resolve this problem, we have refined the project build panel with a special *collaborative view* feature, which is presented in figure 2.

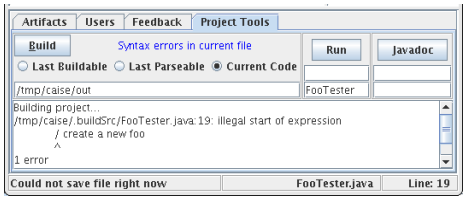


Figure 2: Tools Panel with adjustable levels of project crosstalk.

The view facility within the project tools panel allows compilation to take place from within three different modes: *current*, *last parseable* and *last buildable*. These modes are depicted in figure 3. In current mode, the panel attempts to build the latest version of the code, which will fail if any recent remote changes have broken the build. In last parseable mode, the build only takes into account the last syntactically correct version of each file. This way, if a remote programmer is current editing a file, the changes will only take affect once the code is properly formed. In last buildable mode, the panel will produce an executable based on the last version of the program that had no build errors.

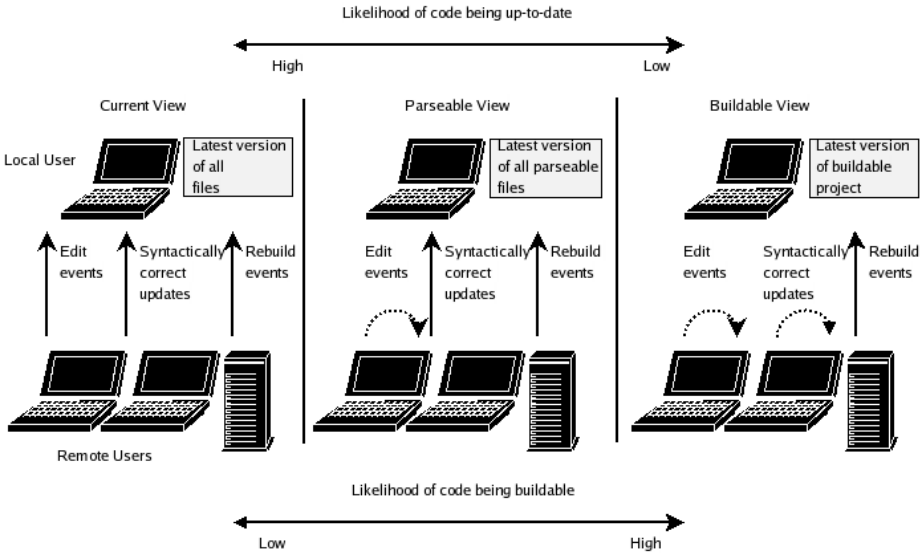


Figure 3: The various modes of collaborative scope when compiling from within a CAISE tool.

This has proved to be a particularly useful feature for real collaborative software engineering, and we suggest it should be employed as a global strategy. It may provide an alternative to partitioning a collaborative group in the case of exceeding a tolerance threshold for remote user activity.

3.2 Implementation Notes

Implementing collaborative tools of any nature is a difficult task. To assist in implementing CSE tools, the CAISE framework provides support for concurrent artifact management and persistence, user management, a distributed event model, and a plug-ins interface for adding new languages and server capabilities.

While the tools presented here appear may appear relatively intuitive to design and implement, the coding effort to build fully-synchronous tools was considerable even with the support of the CAISE infrastructure. An example of the complex implementation process is collaborative undo. To support a realistic software engineering experiment, it was decided that undo needed to be supported within the text editor. Without undo, a mistake could be very costly to correct, which would confound the task completion rates and would also be likely to negatively affect the participants' survey answers. Implementing collaborative undo, where the local user's changes in a file are treated differently from all remote users, is an extremely challenging task within the field of Computer Supported Collaborative Work (CSCW) [22].

4 Evaluation Method

It is a challenging task to design a valid Software Engineering experiment of any kind. For this reason many software engineers leave the task of empirical evaluations to that of other disciplines within Computer Science, disciplines such as Computer-Human Interaction where the number of variables to address are fewer and the difficulty of isolating them is considerably less.

For those willing to design a credible Software Engineering experiment, the first aspect is to determine precisely what it is we are measuring: task completion times, software quality and bug rates, robustness and quality of design, and other subjective measures such as perceived effort and frustration. Following that, we may need to either isolate or explicitly control independent variables such as the scope of the task, participants' familiarity with the tool set, team size and individual roles, and the type of task being performed. Additionally, we need to address potentially confounding factors such as programmer abilities and learning effects. Without isolating the independent variables and addressing confounding factors, a vast number of variables could affect and distort our findings.

Given that is is possible to identify a dependent variable, isolate and control the independent variables, and remove all confounding secondary factors, there are still two considerable issues to consider for software engineering experiments: is the experiment still at a level realistic enough to show an effect that is globally useful; and if an effect is observed, is it possible to claim causality rather than just a correlation.

Once the researcher has convinced himself that such an experiment can be designed to show causality and global significance, he can move on to perhaps the toughest question facing CSE research: what should CSE systems be compared against to give an objective and useful comparison?

4.1 Aim and Purpose

A concern of ours that has also been shown elsewhere is that programmers do not use collaborative systems as much as they can and arguably should [13]. This experiment aims to show that a set of real-time collaborative tools not only provide a more efficient alternative to concurrent program editing with CVS, but participants also prefer using the new tools.

There are many perceived benefits of using CSE tools: faster task completion rates, greater levels of team efficiency, greater understanding of local and remote changes, less or no delay between file updates, fewer or no merge conflicts, and higher levels of communication between programmers. These are all anecdotal claims however; very little empirical research has been conducted in terms of supporting data.

We can only assert a small number of claims in any one trial. Therefore, our primary goal is to illustrate the achievement of faster task completion times using our collaborative tools when compared to an equivalent set of tasks using conventional code repository practices.

As a subjective measure we also aim to assert perceived levels of code change understanding, frustration, success and effort that are more favourable to the collaborative tools than their conventional counterparts.

Finally, we also take the opportunity to ask how much users are likely to use CSE tools in an array of settings. It will be most beneficial to discover if the participants embrace or dismiss the concepts behind the tools. It has been a fear that even though the tools appear superior to us as their designers, 'real' users will not like them regardless of the actual efficiency levels. Empirical evaluations of other tools that seemed a good idea at the time have not always gone according to the researchers' expectations [3].

An auxiliary benefit of this trial is that we can assert that the tools hold up to the fairly intensive test of use by complete outsiders. If the tools reduce task completion times and are favoured by the users, this allows us to confirm that their design, implementation and user interfaces are at least satisfactory in terms of suitability for broad-scale Software Engineering.

4.2 Participants

For this experiment, 12 postgraduate Computer Science students were used, which represented the entire class for an advanced Object Oriented design course. By selecting this class we were confident that we had participants who were interested and experienced in Software Engineering. We also could rely on all participants having at least minimal operational experience in source code repositories and group work. The students possessed grade point averages ranging from satisfactory to excellent. While all participants were male with an even spread of ages from 21 to 30, this is a fairly representative sample of the professional software development population.

After identification of the likely participants, they were left to organise themselves into groups of two. Upon formation of the six groups, most pairs had worked with each other in some Software Engineering context over the last two years.

4.3 Physical Layout

Each evaluation session involved a pair of participants, with the layout of participants and equipment presented in figure 4. As the physical location of participants had the potential to alter the task completion rates, the environment for the evaluations was kept constant for the duration of this experiment. For this experiment, the configuration of the environment was designed to be representative of a typical co-located programming setting.

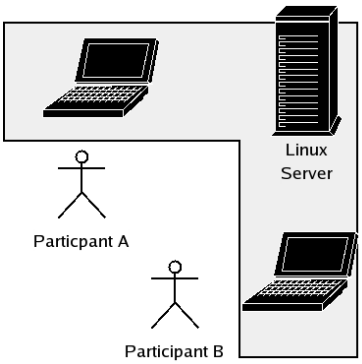


Figure 4: Evaluation layout of experiment.

The participants worked within two meters of each other but the monitors were only in line of sight if they looked over their shoulder to that of the other participant. In this configuration, the participants were able to directly observe and possibly circumvent the activities of each other, but only if they made the conscious effort to draw themselves from their own work. They were at all times able to communicate with each other orally without impairment. Finally, the experiment was held in an isolated office without any risk of interruption or interference.

An interesting point to note is that in a co-located setting such as the one provided for this experiment, it is possible for one participant to observe the work status of the other just by listening for keyboard typing.

4.4 Apparatus

Both the desktop workstations and the CAISE server ran the Fedora Core 3 operating system with the Linux 2.6.9-SMP kernel. The CAISE architecture was compiled with Sun's Java 1.5.0 Standard Edition compiler and executed with the corresponding Hot-spot virtual machine.

The desktop workstations were 32-bit Dells with a single Intel Pentium-4 2.8GHz CPU. The CAISE server was a 64-bit ASUS machine with dual AMD Opteron 2.0 GHz CPUs. All machines had 2 Gb of primary memory.

The desktop workstations hosted the standard Gnome X desktop running at 1600 by 1200 resolution. On each machine ran a local instance of the CAISE server process and the two CAISE tool applications. No other applications ran on the workstations apart from the core Linux services. The server ran without

a display, again running only the core Linux services and the CAISE server process.

The communication medium between the participating machines was a 100 Mbps switched Ethernet network, dedicated for the use of the participants' desktop machines and the CAISE server.

4.5 Procedure

As explained in section 4.1, it is difficult to design an evaluation where a fair comparison of conventional and collaborative tools can be made. In this section we provide details of an experiment that negates as many confounding effects as possible for a set of realistic programming tasks, and isolates the dependent variable of task completion rates for objective measuring.

It is important to emphasise that we are investigating core speed in terms of task completion rates. To do this, we must equalise the effects of programmer ability, physical and mental effort of tasks, task types and scope, and effects of different tools within the evaluation. By isolating these effects we have derived an evaluation that appears somewhat mechanical, but we do in turn achieve a fair measure of the core comparative speeds of the tools. A number of external factors will affect the overall efficiency and effectiveness of collaborative tools, but the experiment will still give us a useful and reliable insight of the CSE tools.

4.5.1 Experimental Design

This section outlines the basic evaluation plan. For the full details of each aspect of the evaluation, please refer to the corresponding related sections within this paper.

Each evaluation session takes approximately one hour per participant group, and only one group is evaluated at a time. The evaluation examined two tool modes: conventional and collaborative, and two work modes: between files changes and within file changes. This equates to four tasks per evaluation session, as presented in table 4.5.5.

All tasks were performed primarily in the editor. The UML diagrammer was available for visualisation of the changing program structure and for user presence awareness. In collaborative mode, the tasks could be performed using the real-time file sharing support of the tools. In conventional tool mode, the participants were able to share and synchronise their files with inbuilt code repository support. The code repository interface was minimal to avoid confounding the experiment, as explained in section 4.5.2. We believe that although we are comparing two different modes of work, conventional versus collaborative, this comparison is fair. Code repositories are the mainstream technology for collaborative software engineering, the only other commercial option today is pair programming with a shared keyboard and display.

For both types of tasks, there was a deliberate and unavoidable conflict between the instructions for both participants. Between-files tasks were such as renaming a method for one participant while the second participant made a new call to the method using the original name. Within-file tasks were such as changing the structure of a complex control statement by one participant while the second participant changed a conditional within the control statement. To

eliminate any variance caused by differing programmer abilities within groups, or by differences in coordinated actions between groups, all tasks were scripted and synchronised for each user.

Each task was hand timed, and students were instructed to work as fast as possible without rushing; this ensured that the participants were focused on completion rates rather than collaboration. To achieve the tasks, however, a degree of collaboration was inevitable. This gives us confidence that the evaluation environment was realistic. The participants had a brief reading period before being timed, where they could clarify any questions related to the task. The participants were not permitted to discuss the task with each other at this stage, however. They could only communicate with each other when completing the task, both face-to-face and by observing the feedback from the tools.

When the inevitable conflict within each task was discovered, normally by completing the scripted instructions and recompiling the code, the participants were then instructed to access an answer sheet which contained the predetermined resolution for the given task. Under normal conditions programmers would discuss and resolve the conflict themselves, but again we needed to remove this factor from the experiment. Timing would stop as soon as the problem was corrected, the code compiled and synchronised, and the program was demonstrated to execute correctly on the workstations of both users.

All evaluation tasks were based on a simple 1000 line graphical Java application. The program consisted of eleven classes within a package that displayed several animated sequences. While the program was relatively trivial, it did contain some fairly complex design idioms such as behavioral, creational and structural design patterns, use of collections classes, graphics code and event-based actions. It was easy for participants to assert that their changes had taken effect; the program at startup would show the animations in their current state which could be immediately verified for correctness. A screen-shot of the program in a typical state is presented in figure 5.

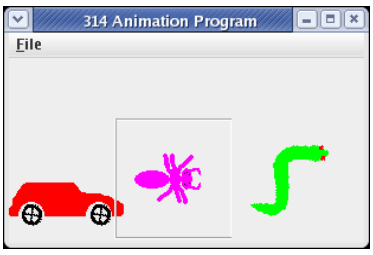


Figure 5: The graphical interface of the program being modified during the evaluation sessions.

Upon completion of each task a survey was given to each participant to answer in private. This allows us to compare the participants' perceived levels of frustration, success and effort for each tool mode and task type. Finally, another survey was completed at the end of each evaluation session, giving us a subjective summary of each users preferences and comments for later comparison.

4.5.2 Supporting a Minimal Code Repository Interface

For this experiment, it was very important to make interactions between the source code repository and the client tools as simple as possible. If the interface to the repository was cumbersome or complicated, it would greatly skew the task completion rates that we are measuring.

We used CVS as the underlying repository, but the users were not aware of this technicality. For our tool, CVS was encapsulated simply as a high level and generic code repository. This is similar to the simple manner in which a Wiki Web supports different versions of files, even though a complicated code repository system is employed on the Wiki server.

Advanced users of CVS and other source code configuration systems will know how to use its features to avoid potential merge conflicts and transactional errors [13]. We are not looking at advanced users however, although we do address this type of user in section 6. Therefore, we assume an average programming ability and only simple use of a code repository system is required, described and supported.

For this evaluation we decided to build code repository support into our own set of tools. Accordingly, when the tools are started in conventional mode, they keep all changes to files isolated from other users and changes between users can be synchronised through a code repository menu. This menu is presented in figure 6.

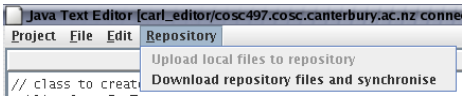


Figure 6: The syntax directed CVS interface.

Not only did we want the code repository to be a simple as possible to use, and accordingly as fast as possible, we also wanted to ensure that our participants did not make errors when synchronising their source files. We are well aware that many users struggle with systems such as CVS; typical problems include forgetting to download and work on the latest version of the repository, forgetting to upload new or modified files back to the repository, and *checking-in* a subset of files that build locally but will break the repository's version of the program.

Given the collaboration-intense nature of the evaluation tasks, we were certain that mistakes were likely if a conventional code repository interface was provided. Accordingly, we developed a *syntax directed* interface to the repository where only valid repository actions could be made given a set of local files and a central code repository. Participants can only upload their files if they have modified the latest version from the code repository. Similarly, the option to download newer files from the repository is only available if the participant's local version is out-of-date. If the download option is available, a merge of all newer repository files and the local copy is made automatically upon completion of the download. This allows the user to upload the newly formed set of local files once all required changes have been made, if any, without risking uploading an out-of-date set of files.

Another aspect of the code repository interface is that it performs the upload and download actions as a batch where all files are included for transfer. This way we avoid problems where files are accidentally excluded resulting in skewed sets of file between other users and the code repository. Participants only have to be aware that they need to upload all their changes before finishing the task. This ensures that the two users' sets of files will be synchronised before completion of the evaluation task; the second user will be forced to perform a download and merge before the upload option becomes available for checking back into the repository.

It took many prototype designs and pilot studies to create the code repository interface presented here. We believe that this type of syntax directed interface is a highly appropriate mechanism to support making a comparison between conventional and collaborative software engineering. We are able to compare the essential differences between the two modes of work, but we no longer have to address factors such as the time it can take to type in code repository commands or to navigate through a cumbersome repository interface, getting the code repository commands right for the current state of the local files and the global repository, and the imbalance of repository experience levels between participants.

As an aside, many users commented on how intuitive and easy-to-use the code repository interface was, and they would like the same interface on all of their usual Software Engineering tools.

4.5.3 Tool Modes

As this experiment compared conventional versus collaborative software engineering task completion rates, we required our set of tools to run in both collaborative and conventional modes. In this manner, as long as the code repository interface is minimal, there should be no confounding factors in terms of tool type—the participants can use the same tool for both tasks, with very little practical difference between the two tool modes.

When the tools were operating in collaborative mode, a central server was responsible for supporting communication between all participating tools and for keeping code synchronised in real time between participants. In conventional mode, the central server was not employed; a local instance of the server process was used to maintain code at the scope of each individual workstation. To synchronise the code modifications between participants, the code repository interface would access a CVS server on a local network file system partition.

The learning effects of individual tool modes is addressed in section 4.5.5. Additionally, to keep the workstation memory loads constant between tool modes, a local copy of the server process ran on each workstation in collaborative mode, albeit redundant.

4.5.4 Task Types

This evaluation assessed two types of task completion rates. One was for task completion rates of between files changes, the other was within files changes. Collaborative software engineering is normally a combination of both types of programmer/file interaction, but we need to treat both cases separately in order to remove any effect of interaction on task completion rates.

All tasks within both sets were designed to generate some sort of conflict between the two participants. For between-files tasks, a *transactional* conflict would occur, meaning there is a problem with the program semantics as a result of the concurrent modifications. A transactional conflict is one where the syntax of the changes is legal, but a semantic error would result once both participants' modifications were synchronised. For example, one user might rename a method while the second participant would make a new call to the method by the original name. Only when the files are synchronised and the resulting code is rebuilt will the error be exposed.

For between-files tasks, a *merge* conflict would result after each participant had made their change and synchronised their code, meaning that there is a problem with the program's syntax as a result of the concurrent modifications. A merge conflict results from overlapping modifications to separate local copies of a source file; when the code repository system attempts to synchronise the changes from multiple users it fails because of there is no deterministic way of forming a final, conflict-free version of the file.

As an example of a merge conflict, one participant could be editing a sequence of statements within a method so that instead of evaluating several complicated conditionals, the code is refactored as a more comprehensible switch/select block. At the same time, the other participant might be editing a second copy of the original file so that one of the conditionals is simplified syntactically. In this case, most code repository systems would give a merge conflict error where it is up to the participants to resolve the merge conflict manually and resubmit the final version to the code repository.

4.5.5 Order of Groups and Tasks

The order of groups in which the evaluations were held and the order of tasks that each group performed are presented in table 4.5.5 (Cv stands for conventional mode, Cb stands for collaborative mode. T1 and T3 are between-files tasks, T2 and T4 are within-files tasks). Careful consideration was given to the design of task ordering between groups; the main objective was to negate or minimise any learning effect of tool mode and task type.

Group	Task Configuration				Order			
1	CvT1	CvT2	CbT3	CbT4	1	2	3	4
2	CvT1	CvT2	CbT3	CbT4	4	3	2	1
3	CvT1	CvT2	CbT3	CbT4	1	3	2	4
4	CbT1	CbT2	CvT3	CvT4	4	2	3	1
5	CbT1	CbT2	CvT3	CvT4	3	2	4	1
6	CbT1	CbT2	CvT3	CvT4	1	4	2	3

Table 1: Task types, tool modes and order of tasks.

We used each pair of participants for both the treatment and the control group. To make this possible we employed separate yet similar tasks for each tool mode; this is the reason why there are two tasks for each task type. By using the each group as a treatment and control, we negate any imbalance between individual groups. If one group is exceptionally good or bad at a given task, they are likely to produce the same result for both tool modes.

To reduce the risk of a learning affect on task type or tool mode, each group had a different order of task type and tool mode. Group one, for example, first

did both tasks in conventional mode and then collaborative mode. Group two did both tasks in collaborative mode first, followed by conventional mode. From table 4.5.5 we can also see that the task modes were also alternated between groups. If there was any learning effect from task type or tool mode, it was likely to be countered by the nature of the group and task assignments.

Since participants acted as both the control and treatment group, the only other confounding factor could come from differences within the sets of tasks. While it may at first seem relatively simple to create two similar tasks for a each task type, in practice this was quite challenging to achieve. We needed to ensure that the tasks were distinct to reduce any learning affect yet nearly identical in terms of syntax, semantics, typing effort and conflict resolution actions to ensure that the tasks were objectively comparable.

In section 4.5.9 we show that from analysis of the experiment results there was no significant difference between any pair of task types for each tool mode. Again, if there was a significant difference within a set of tasks of the same type, due to the design of the group and task order, the impact would be largely negated.

4.5.6 Training Manual

Participants were given an intense 30 minute training session prior to completion of the evaluation tasks. It is hoped that by giving a thorough training, learning effects of tools and task types were minimised. To assist the training period, a training manual was given to each participant a few days prior to the evaluation session. This gave the participant a chance to gain an overview of the tools and tasks, and allowed him or her to prepare questions for the training session.

The training manual provided the participants with an overview of the code repository system, how to operate the code repository within the evaluation tools, how the real time editor and awareness support components operate, and how the basic editing and compiling functionality works for both tool modes, such as cut, copy, paste, undo, compile and run. An excerpt from the training manual is given in appendix A.1. The appendix also gives details on how to obtain a full electronic copy of the training manual.

Training Tasks Within the training manual there are four mechanically-scripted tasks to complete. Two of the tasks are conflict-free, the remaining two contain inevitable conflicts. Two of the tasks involve within files changes, the remaining two involve between files changes. The four tasks are performed by each pair of users firstly using the tools in conventional mode, and then again with the tools in collaborative mode. An excerpt from the training tasks sheet is given in appendix A.2.

Answer Sheet Each conflicting task within the training session has a prescribed resolution. When the participants encounter a conflict, they are instructed to refer to the training tasks answer sheet for the correct resolution. The answer sheet simply details which parts of the code need to be replaced, and what these lines of code need to be replaced with. Upon correct resolution of the conflicting code changes, the program should compile again, and the task is then considered to be complete. An excerpt of the training tasks answer sheet is also given in the appendix.

4.5.7 Evaluation Tasks

The evaluation tasks were again mechanically scripted for each participant, with check-boxes on the manuscripts to help prevent participants from skipping instructions or performing operations in an incorrect order. As in the training tasks, an answer sheet was provided to resolve the resultant conflict from the two sets of changes. An excerpt of the evaluation tasks sheet and answers is given in appendix A.3.

Each participant worked as fast as they could on their set of instructions, and the first group member to complete his work, including recompiling the code and verifying that the application still worked properly, could submit his work to the code repository first and not have to deal with any potential transactional or merge conflicts. In all cases, the participant who finished his tasks second had the task of correcting the now exposed conflict. For the tasks that were performed in collaborative mode, the issue of which participant did the code correction was determined by whoever discovered the conflict.

4.5.8 User Survey

A survey was given to each participant to complete in private at end of each task. The aim of the survey was to provide a comparison between the evaluation tool for the current mode and given task type against any previous experience of collaborative Software Engineering tools.

For this survey we used NASA-TLX questions to determine and compare the perceived effort, success and frustration levels of each participant [14]. By using the standard NASA-TLX questions, we make our results available for comparison against any other related studies that use the same survey technique. We also added some additional subjective questions related to the understanding of code changes and the perceived ease of file control.

A survey was also given at end of each session. This was purely for feedback on the underlying concepts of our system, such as did the participant like the concept of real time code sharing and editing, and would the user consider using such systems if they were made available.

The questions for both surveys are given in the results section, and the full surveys are given in appendix A.4. For both types of survey we used a 20 point Likert scale in all questions; this is conventional for most NASA-TLX surveys as it gives a far more accurate and robust representation than conventional five point scales.

4.5.9 Statistical Validity

We need to ensure that any use of statistics is valid and justifiable before we analyse the data and report the results. Aspects to consider when looking at the statistical validity of empirical software engineering tasks include the choice of statistical test, design of the experiment to eliminate confounding factors, and post-evaluation analysis of data to ensure it fits with test.

Choice of Test We selected one-way analysis of variance (ANOVA) for this evaluation. We are testing to see if there is any statistically significant difference between the sample means, where separate tests are conducted for within files

and between files tasks. In the case of this evaluation there were only two means to compare, so a two-sample t-test could have been used to give identical results.

We did not investigate interactions (two-way ANOVA) of tool mode and task type. This would be an interesting aspect to explore, but it is not the focus of this study. While we can not rule out that there could be some interaction between the task type and tool mode, our study focused on specifically isolating each task type.

From literature related to the use of empirical statistical analysis [15], it is safe to assert a statistically significant, valid and meaningful difference between two means if:

- The power of the test is not too high. A high power test is susceptible to asserting that a negligible difference between two means is statistically significant.
- All samples are a simple random survey (SRS) of the population, where the population follows a normal or near-normal distribution. This also implies that the samples should follow normal or near-normal distributions with similar standard deviations to each other.
- The sample sizes are the same or similar to each other.
- The measures of both samples are independent of each other.
- There is no bias in the experimental design.

A related point to raise is that of a low powered test. If the aim of the experiment is to assert that two or more means are the same, then a low power test should not be used. A low power test which asserts that means are no different risks a considerable possibility of being incorrect.

We believe that the evaluation has not breached any of the above guidelines, and therefore the results of the statistical tests are significant, uncompromised, and genuinely useful and applicable to the field of CSE research. Justification of this claim is provided in the remainder of this section.

Design of Trial A common criticism of statistical tests is that unless there is a large number of observed values, the results are not valid due to the low statistical power of the test. This criticism is only valid when asserting similarities between a set of means, not differences. For the evaluation presented in this paper, we are only interested in finding a statistically significant difference between the task completion rates for two tool modes; if any difference is found then it is valid.

To assert a difference is a challenging task, however. We require means quite distant from each other, and standard deviations small enough that they do not overlap. To achieve both of these characteristics from the data we would normally require a large, high power sample to reduce the standard deviation size, or data that genuinely are from populations with well separated means.

For this evaluation the sample sizes were all the same within each statistical test. Additionally, we are comfortable claiming that the pool of participants was representative of the population, and can be considered as a SRS. This is discussed further in section 6.

If we tested the two tool modes on the entire population we would expect an approximately normal distribution of completion rates—most users would complete the tasks near the population mean, with a decreasing number of outliers either side of the mean. In other words, we foresee no skew or flatly uniform distribution if the entire population were to be sampled.

We can safely assert that the task completion rates taken from both samples were independent of each other. In the case of this evaluation, the two samples actually consisted of the same set of participants, but being examined under different tool modes. As long as the learning effect was negligible, then independent measures could be assumed.

As discussed in previous sections, we have taken strong steps to eliminate or reduce any bias within the experimental design. Potential sources of bias are learning effects on tool mode and task type, and we have taken steps to eliminate this. We have also taken steps to remove any other confounding factors such as programmer ability and scope of tasks by isolating and mechanising the experiment as much as possible. A classic source of bias in evaluations where students are enlisted as participants is that of self-selected data. We eliminated this risk by ensuring that the entire class took part in the evaluation, not just the students who showed interest.

Post Data Analysis After completing the evaluations and collecting the raw task completion rates and survey responses, it was possible to verify our assertions of normally distributed samples and equivalent sample standard deviations.

The first step in any post-data analysis is to plot the results and confirm that the distribution looks normal and the standard deviations are also of approximately the correct magnitude. In the case of this experiment the data for both the objective measures and the subjective measures appeared satisfactory.

To formally test for equivalence between standard deviations, the rule

$$max(s.d.) <= 2 \times min(s.d.)$$

is often followed [18]. All our statistical tests conform to this rule for task completion rate comparisons. As we wanted to test for significant differences within the survey questions, we also checked our survey results. All but three of the twelve survey tests for statistical differences passed this rule. For the three tests that failed in terms of having equivalence, the p values were all so small that it is safe to assume that the result were still significant in determining a statistical difference [18].

A final concern that we could dismiss by statistical investigation was that of unfair variance within tasks of the same type. As the experimental design required two unique tasks for each type, we need to ensure that the completion times were similar for each task within both tool modes. If no significant difference is found in times between both tasks within each task set, this eliminates any speculation of a confounded experiment due to non-equivalent tasks. While any disparity between tasks is negated by the order of the groups and tasks, it is beneficial to assert that there is no disparity in the first instance.

Our hypothesis is that there is no difference between the means of the groups that completed the two different tasks for each given task type and tool mode. For all ANOVA tests, we compute the F statistic which is the ratio of variance

between and within groups. A low F statistic implies that there is a lot of variance between groups compared to the within groups variance, meaning it is likely any difference between the means is due only to chance. When testing the F statistic, we compare this to the $F(I-1, N-I)$ distribution, where I is the count of groups and N is the count of all samples taken. We reject the null hypothesis that the means are equal if the calculated F test statistic is larger than the critical value of the F distribution for the corresponding degrees of freedom (I and N) and confidence level. In other words, we check the allowable variance against the actual variance found in the given trial. If the variance ratio is below the critical point for statistical difference, we can not reject the null hypothesis of equal means.

If we inspect the resultant p values from an ANOVA report, we reveal the actual probability that the difference between the sample means has occurred purely by chance. When we say, for example, that we have a computed $F_{1,4}$ statistic of 0.13 with a p value of 0.74, this implies that for 74 trials out of every 100, the difference occurred purely by chance. In this case, we have no evidence to reject the null hypothesis that the sample means are the same. This high likelihood of detecting a difference only by chance reflects two distributions that are centered around a similar mean with distributions that overlap considerably. As an alternative view, the F test statistic of 0.13 is considerably lower than the critical value of 7.71 for $F_{1,4}$ at the 5% significance level.

After performing the test we were not able to show differences between any of the means within a set of tasks for a given tool mode. In collaborative mode, the between files test gave $F_{1,4}$=1.81, p=0.25 and the within files test gave $F_{1,4}$=0.13, p=0.74. In conventional mode, the between files test gave $F_{1,4}$=4.69, p=0.10 and the within files test gave $F_{1,4}$=0.37, p=0.58. This gives evidence that there may not be any difference between tasks for a given task type and tool mode, as we propose, but to claim outright no significant difference with a test of such a low power would be considered unwise.

4.6 Summary

The experimental design took a considerable effort design, with duties including precise and unambiguous training manual and task wording, order of task design, and verification of statistical validity. We were also required to derive two similar yet distinct conflicting tasks for both within files and between files experiments, and numerous pilots of the evaluation were undertaken to ensure that the individual sessions ran smoothly. Accordingly, it is envisaged that this experimental design can be replicated to save the time of others, perhaps even using the same set of tasks. Additionally, by using the same experimental design and set of tasks in other studies, an objective comparison between different tools can be made.

5 Evaluation Results

This section provides details on the findings of the evaluation for the six groups of participants. The results are discussed further in section 7.

5.1 Task Completion Times

The task completion times for the tools in collaborative mode were at least twice as fast as the times recorded for the tools in conventional mode. The comparative differences are presented in figure 7. For within file tasks the difference was highly significant, ($F_{1,10} = 38.3$, $p<0.01$) as were the between file differences ($F_{1,10} = 34.2$, $p<0.01$). These significance levels give us confidence that the results were not obtained by chance; we expect to achieve the same result for 99.9% of trials that repeat this experiment.

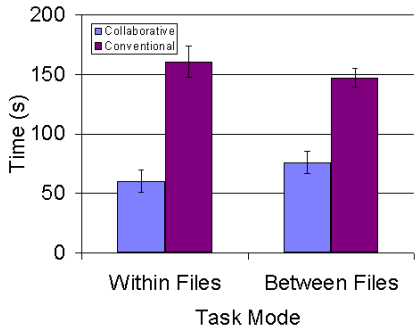


Figure 7: Mean task completion times. Error bars show the mean ± one standard error.

5.2 Subjective Assessment

Table 5.2 presents the findings of the survey given at the end of each task within the evaluation sessions. As mentioned previously, the survey is based on the NASA-TXL index with a 20 point Likert scale. From the table we see that for both task modes, participants felt strongly that they understood the changes of others better, and it was markedly easier to control source files using the tools in collaborative mode.

	NASA Task Load Index: Within, Between					
	Understanding own changes	Understanding others' changes	Ease of File Control	Perceived Effort	Perceived Success	Perceived Frustration
Collaborative:						
Mean	14.7, 12.1	18.8, 9.2	16.3, 15.9	3.9, 2.9	17.9, 16.3	3.7, 4.3
(s.d.)	(4.9, 4.2)	(1.4, 5.9)	(3.5, 3.5)	(3.5, 2.5)	(2.2, 2.5)	(2.4, 2.8)
Conventional:						
Mean	9.0, 8.8	4.5, 1.8	8.4, 7.6	5.3, 7.5	15.5, 14.4	6.1, 8.3
(s.d.)	(4.5, 6.0)	(4.3, 1.4)	(4.3, 5.6)	(3.8, 5.5)	(3.6, 4.3)	(4.0, 5.7)
***<.01; *<.05	***,–	***,***	***,***	–,*	–,–	–,*

Table 2: Summary of the subjective measures for tasks: NASA-TLX workload ratings. Possible values range from 1 (low) to 20 (high).

For perceived frustration, perceived effort and awareness of local changes, there was a statistically significant difference between the mean response in one of the two task modes in favour of collaborative mode. For the remaining task

mode in each survey question, the difference was still favourable towards collaborative mode, but the difference was not statistically significant. For the perceived success survey question, neither task mode gave a significantly difference in mean response, although the participants again showed a lenience towards the tools when running in collaborative mode.

5.3 User Preferences

Table 5.3 presents the findings of the survey given at the end of each evaluation session. This survey focused on general user preferences using a 20 point Likert scale. The questions within this survey are also presented in table 5.3.

Order	Question	Response: mean (s.d.)
1	In a collaborative, distributed setting, how useful do you think this type of system will be?	15.7 (2.1)
2	In a collaborative, co-located setting, how useful do you think this type of system will be?	15.8 (1.9)
3	How much does it help to have the source code shared and managed for you?	16.4 (2.2)
4	How often would you like to work on collaborative tasks with a system such as this (a system that updates and shares source files in real time)?	14.3 (2.2)
5	How useful did you find the ability to know what the current global state of the project is?	14.8 (3.5)
6	How adequately was the awareness support provided (such as user location feedback)?	13.0 (4.1)

Table 3: Summary of the subjective measures for overall preference. Possible values range from 1 (low) to 20 (high).

The results of the user preferences survey was encouraging—all responses ranged from positive to extremely positive. The participants foresee the collaborative tools as useful in both co-located and distributed settings, they find the real time synchronisation of code helpful, the feedback support was also perceived as useful, and they would use CSE tools such as the those used in the evaluation often if made available.

5.4 User Comments

Instances of recurring comments made during and after the trials are listed in table 5.4. Of the positive comments we conclude that all users enjoyed using the system, and they claim that they would use it for most situations given the opportunity. They also stated that they liked having the source code managed for most tasks. These comments are corroborated by the results of the user preferences survey reported in section 5.3.

Of comments to help improve the system, a private work facility is now at the top of our list for future work; we had considered the idea before and users appear to be asking for it as well. The remaining comments for improvement were all related to usability issues we know that we must address as soon as possible.

Type	Comment
✓	"The system made coding more enjoyable."
✓	"I liked the concept of real time development."
✓	"The collaborative [user] tree was really helpful."
✗	"The [editor] lag was a bit annoying."
✗	"A private work area is needed for offline [development] spikes."
✗	"The editor needs tele-scrollbars to give a better indication of where other users are within the same file."

Table 4: User comments.

6 Threats to Validity

A threat to validity that all empirical researchers face is that of the appropriate use of statistical tests. As discussed in section 4.5.9, we believe that our design of experiment and use of statistics is valid and meaningful.

Another aspect that is open for discussion for many evaluations is that of assuming the trial group is in fact a SRS of the global population. This judgement can be made by software engineers, statisticians, or perhaps more suitably both groups. A statistical purist might argue that a SRS has not been made in the case of this evaluation, as we have taken the entire population of the class. Alternatively, a software engineer can argue that this class is a SRS from the population of typical every-day software engineers. Typical programmers are hard to define, but experienced and competent Software Engineering students are probably a suitable average.

Perhaps the threat to validity with the most impact within this evaluation is that of the source code repository interface. If the interface is too simplistic, advanced repository users will not be able to use the repository in their usual manner to avoid getting into programming deadlocks and conflicts when using the tools in conventional mode. For the set of participants used in this trial, not one participant mentioned that the interface appeared limited, which suggests that only highly experience code repository users know how to use them to avoid concurrency issues as shown elsewhere [13]. We are still confident, however, that the experiment was indicative of activity commonly associated with code repository usage, particularly by non-expert users.

On a similar topic, it is conceivable that the user survey results between pairs might have unfairly high variance due to the imbalance of work—as discussed in section 4 the first person to check his or her files back into the repository has less work to do than the remaining participant. We envisage that the combined opinions of each group should give a fair comparison against the mean efforts in collaborative mode. It does imply, however, that the tests for differences between means are potentially less significant than if the effort levels between participants were perfectly balanced.

A constant factor that we have kept in mind since the idea of performing user evaluations on our set of tools was that of keeping the experiments realistic yet measurable. If the experiments are too sterile then we risk having results that are valid but not genuinely useful in a global context. Unfortunately, the tasks do need to be reasonably sterile to enable them to be repeatable and free from confounding factors. While we have designed the tasks as somewhat

artificial, particularly from the aspects of scripted participant instructions and conflict resolution, we believe that the evaluation is a reasonable approximation of tasks and conflicts that will be encountered in everyday Software Engineering.

7 Discussion

The results obtained for task completion rates and subjective measures were surprisingly good considering that no attention had been paid to making the tools particularly user friendly or refined. While we were very confident that there would be some difference between the two tool modes in favour of collaboration, we were surprised that the differences were so large. More pleasing, however, was the subjective results which showed that users liked using the system and agreed with our own perceived benefits to software engineering. It was always a concern that even though the users could perform the tasks faster, they did not like using the tools in collaborative mode.

While the evaluation tasks involved at least a degree of collaboration between users, the tasks were not designed specifically in favour of a highly collaborative approach. Therefore, for tasks that are highly collaborative, such as debugging or demonstrating new ideas, we have reason to believe that the tools in collaborative mode would perform even better than in this experiment. Similarly, as the users only had about ten minutes worth of training in collaborative tool mode, we observed that the collaborative features of the tools were not used to their fullest potential. Given more experienced users, it is highly possible that the task completion rates could have been improved upon, and the feedback on the collaborative mode of work might have been even more positive.

When referring back to the data presented in figure 7, there was a considerably larger gain for collaborative within files tasks than collaborative between file tasks. A likely explanation for this is that it is not possible to avoid transactional conflicts in between files tasks as it is to avoid merge conflicts during within file tasks. Programmers still have to discover the transactional error and then correct it for between file tasks, whereas with within files tasks they can detect the potential conflict and avoid it altogether. Regardless of the relative difference between the two tool modes, between files tasks are still a lot faster in collaborative mode than conventional mode because the error is detected immediately, not after a file merge and rebuild.

An interesting observation during the experiments was that when participants did not stop and talk with each other in collaborative mode for within files tasks, they still managed to accomplish their code changes without noticeable hindrance. They simply engaged in a brief 'editing war' where even though their changes were being interrupted, both users very soon had their code changes in place. Under normal circumstances we would expect users to slow down and discuss collaborative edits that occur in the same region of code, but some participants in this experiment were highly task oriented due to the nature of the evaluation.

We feel confident that our results can be used beyond the scope of this evaluation, within reason. Given larger numbers of users within a group, a wider range of users, and different types and sizes of tasks, we believe that our tools would give similar results to those reported in this paper. Certainly some common tasks, such as algorithm design and development, should be performed

in a private work area, but the majority of Software Engineering tasks are likely candidates for CSE tools such as ours. As the number of users in a group increases, so too does the amount of collaborative user activity, but the same problem also faces conventional tools and code repository systems.

In summary, all of the measurable aspects of this evaluation produced pleasing results, which assists us in confirming both the quality of the current set of CAISE tools as well as the principles of the underlying architecture. Additionally, we have also engaged in other forms of evaluations for the set of tools currently under investigation. For example, we recently proposed a set of heuristic evaluations for CSE tools that promote their continual improvement during the entire development life-cycle [6]. Accordingly, the set of CSE tools presented in this paper are constantly refined through such heuristic evaluations, which allows us to anecdotally confirm the solidity of their design.

8 Conclusions and Future Work

Synchronous Software Engineering, including associated tools, is an area of active research at present, with many new tools being introduced both throughout the research field and the software engineering industry.

We have developed one of the first sets of general-purpose tools to engineer software collaboratively and in real time. Many questions arise from these tools and the associated modes of work, such as do the tools really work in practice, and will programmers want to develop software collaboratively given the technologies to do so. It is very important to start objectively evaluating such tools; there have been many papers of tools that appeared as good ideas in the research lab but when evaluated failed to reach the expectations of both the researchers and test users.

Through our evaluations we have shown examples where our set of CSE tools not only significantly outperform their conventional counterparts, but users prefer using them, their perceived success is higher, and their perceived effort and frustration levels are lower. Our results strongly suggest that collaborative tools such as text editors can improve the productivity of software development. Subjective results also suggest that providing users with a constantly updated global project state appears to help developers rather than hinder. Most other aspects of participant feedback were highly positive as well.

In summary, we can now confirm through empirical results that our anecdotal assumption of moving software engineering tools into the realm of computer supported collaborative work has real benefits in terms of task completion rates, and other perceived benefits as well. We have also shown that our tools stand up to testing with users that have had no previous exposure or experience to them, even when completing considerably comprehensive tasks within a non-trivial application. From the results of this statistical evaluation, previous positive feedback from associated researchers and software engineers, and continual refinement from heuristic evaluations, we are strongly encouraged to continue further research and development of collaborative tools for software engineering.

This evaluation was restricted to investigating the effects of collaboration on core task completion rates where all other software engineering factors were controlled. We encourage other researchers to repeat this experiment on their tools for comparison. For our research, an important further step is to investi-

gate how users interact with each other and the collaborative tools given more complex and open-ended sets of development tasks.

We also advocate investigations into actions of participants at a fine-grain level during collaborative tasks over long periods of development [6]. Given the positive user feedback from this evaluation, and the reliability of the latest version of CSE tools, a longitudinal study of collaborative development behaviour will be the basis of another future study.

References

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

[2] Marko Boger, Thorsten Sturm, Erich Schildhauer, and Elizabeth Graham. *Poseidon for UML User Guide*, 2002. URL `http://www.gentleware.com/support/documentation.php4`.

[3] R. P. Carasik and C. E. Grantham. A Case Study of CSCW in a Dispersed Organization. In *CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 61–66, New York, NY, USA, 1988. ACM Press. ISBN 0-201-14237-6.

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

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

[6] Carl Cook and Neville Churcher. Modelling and Measuring Collaborative Software Engineering. In Vladimir Estivill-Castro, editor, *Proceedings of ACSC2005: Twenty-Eighth Australasian Computer Science Conference*, volume 38 of *Conferences in Research and Practice in Information Technology*, pages 267–277, Newcastle, Australia, January 2005. ACS. 25% acceptance rate.

[7] Carl Cook and Neville Churcher. Collaborative Software Engineering Tools. Technical Report TR-COSC 05/05, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, August 2005.

[8] Carl Cook, Neville Churcher, and Warwick Irwin. Towards Synchronous Collaborative Software Engineering. In *Proceedings of the Eleventh Asia-Pacific Software Engineering Conference*, pages 230–239, Busan, Korea, December 2004. IEEE Computer Society.

[9] Borland Software Corporation. What's New In Borland JBuilder 2005. White Paper, September 2004. URL `??`

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

[11] Jon Froehlich and Paul Dourish. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In *6th International Conference on Software Engineering (ICSE'04)*, pages 387–396, Edinburgh, Scotland, United Kingdom, May 2004. IEEE.

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

[13] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group Awareness in Distributed Software Development. In *CSCW '04: Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 72–81, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-810-5. doi: http://doi.acm.org/10.1145/1031607.1031621.

[14] S. G. Hart and L.E. Staveland. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In P.A. Hancock and N. Meshkati, editors, *Human Mental Workload*, pages 139–183. Elsevier Science, 1998.

[15] C. Helberg. Pitfalls of Data Analysis (or how to avoid lies and damned lies). In *Third International Applied Statistics in Industry Conference*, Dallas, Texas, U.S.A.,, June 1995. URL `my.execpc.com/ helberg/pitfalls`.

[16] Sun Microsystems Incorporated. Sun Java Studio Enterprise Edition. Datasheet, July 2005. URL `www.sun.com/software/products/jsenterprise/ds_jsenterprise.pdf`.

[17] Scott Lewis. Eclipse Communication Framework. Internet Homepage, April 2005. URL `http://www.eclipse.org/ecf/goals.html`.

[18] David S. Moore and George P. McCabe. *Introduction to the Practice of Statistics*. W H Freeman and Company, New York, 2nd edition, 1993. ISBN 0-7167-2250-X.

[19] Martin Ott, Martin Pittenauer, and Dominik Wagner. SubEthaEdit. Web Site, July 2005. URL `www.codingmonkeys.de/subethaedit/collaborate.html`.

[20] Michael Reeves and Jihan Zhu. Moomba A Collaborative Environment for Supporting Distributed Extreme Programming in Global Software Development. In Jutta Eckstein and Hubert Baumeister, editors, *Lecture Notes in Computer Science*, volume 3092, pages 38–50. Springer-Verlag, January 2004.

[21] Anita Sarma and Andr van der Hoek. Palantr: Coordinating Distributed Workspaces. In *26th Annual International Computer Software and Applications Conference*, Oxford, England, August 2002. IEEE.

[22] Chengzheng Sun. Undo as Concurrent Inverse in Group Editors. *ACM Transactions on Computer-Human Interaction*, 9(4):309–361, 2002. ISSN 1073-0516.

A Evaluation Documents

The full task sheet, training manual, answer sheets and participant surveys are available for download from `www.cosc.canterbury.ac.nz/clc/cse`.

A.1 Training Manual

Excerpt from the introduction

Basic Training Manual for the SEVG's Collaborative Software Engineering Tools Evaluation

Preliminaries:

- The upcoming evaluation is to test our software, not your ability to program.
  - We are interested in the task completion rates and your feedback related to the tools.
  - The tasks are very simple, and require no actual software engineering experience or ability. Each task will be mechanically scripted for each user in both the training and evaluation session.
  - There are many aspects of the tools and underlying framework that need to be evaluated. For this evaluation, however, we only have time to look at task completion rates for specific types of scripted tasks. In future evaluations we will address usability issues, programmer comprehension, levels of inter-programmer communication and interaction, levels of software quality, etc.

Training Session:

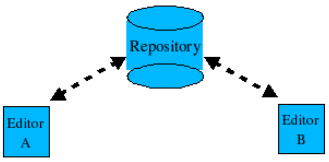
- There will be a thirty minute training period prior to starting the evaluation tasks
- The tasks in both the training and evaluation sessions should be approached in a mechanical manner,

Excerpt from the CVS interface section

Conventional mode and the code repository

When the tools are running in conventional mode, source files are only shared between the tools on your desktop. To share your source files with your other team member, you will need to use the code repository system (which is built into the text editor).

The following is a brief description of the code repository system, but *you don't need to memorise this or fully understand how it works in detail.* **We will shortly go through some examples which use the code repository, which should provide you with the basic working knowledge to operate the repository for this evaluation.**



A.2 Training Tasks

Excerpt from a conventional between files training task

Task 1: *Working on two separate files*

User A opens the file `Foo.java`
User B opens the file `FooTester.java`
User A:

- Renames the method `Foo.saySomething()` to `Foo.sayAnything()`
- Replaces the method call to `saySomething()` with `sayAnything()` from within the method `sayHello()`.
- User A then selects Upload, posting the final version of the file back to the repository. He then closes his file.

User B:

- Add a new call to `foo.saySomething("hiya")` from `FooTester.test()`.

User B now can not select Upload, so selects Download instead

- This downloads the latest version of the files that the other user has been working on

User B compiles his code:

- A transactional conflict is discovered.

User B then consults the answer sheet (task 3) to find out the resolution for the problem.
When user B is happy with the code again, he selects Upload and closes his file.
At this point, User A could select Download to bring his files back in sync with the latest version.

Excerpt from the training tasks answer sheet

Task 1

In `FooTester.test()`:

rename:

```
foo.saySomething("hiya");
```

to:

```
foo.sayAnything("hiya");
```

Task 2

In `FooTester.test()`:

The conflicting statement should be merged as:

```
Foo myFoo = new Foo(Foo.MAORI);
```

A.3 Evaluation Tasks

Excerpt from a conventional within files evaluation task

Task 1 [Conventional]

- User A
  - Open the file `AnimatedSprite.java` and locate the method `AnimatedSprite.advanceImage()`
  - In the method `advanceImage()`, change the conditional:
    `iconList.size() > 0`
    to:
    `!iconList.isEmpty()`

❒ *Build and run your code to make sure the program still works.*
❒ *When ready, upload your code to the repository.*
- *If the upload option is not available, this means that your partner has already uploaded newer versions of the source files to the repository. In this case:*
  ❒ *Select download to obtain the latest version of the source files (this will automatically update your copies of the source files with the latest versions held in the repository)*
  - *If you encounter merge conflicts during the download:*
    ❒ *Fix the merge conflicts by referring to the help sheet*
  ❒ *Compile your code again to make sure that there are no problems*
  - *If there are build problems:*
    ❒ *Fix the problem by referring to the help sheet*
  ❒ *Upload your code to the repository*
  ❒ *Ask your partner to download the repository again*

❒ *Close the file within the editor once finished*

Excerpt from the evaluation tasks answer sheet

Task 2

In `AnimationApp.makeAnt()`:

The method call should be changed to:

```
ant.getSimpleSprite().setBoxed(true);
```

Task 3

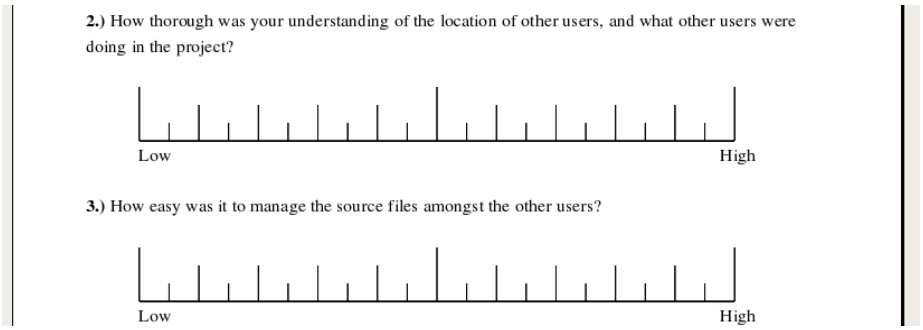
In `AnimatedSprite.advanceImage()`:

The conflicting statements should be merged as:

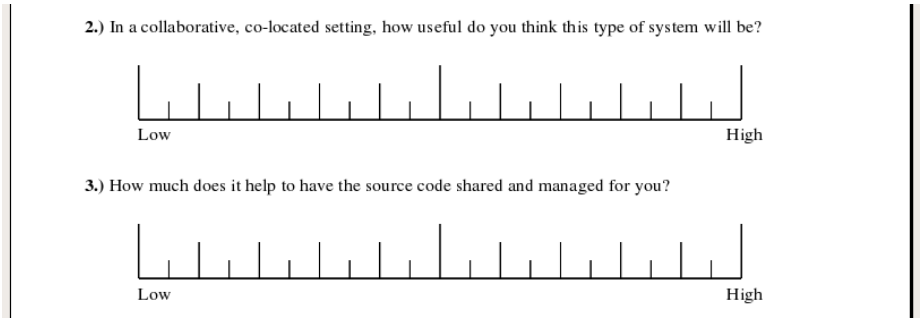
```
if (!iconIterator.hasNext())
     iconIterator = iconList.listIterator();
```

A.4 Participant Surveys

Excerpt from the end of task survey



Excerpt from the end of session survey



B Source Code

The Java source code for both the training and the evaluation applications are also available for download from `www.cosc.canterbury.ac.nz/clc/cse`. This includes the images used for the animations.