Collaborative Software Engineering: A Survey

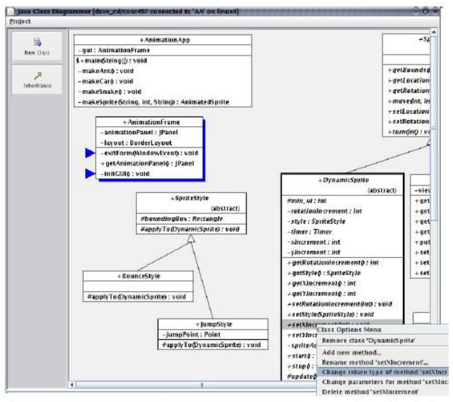
Agam Brahma

November 21, 2006

Abstract

This paper surveys recent work in the field of collaborative software engineering and relates it to concepts discussed in the course 'Collaborative Systems', with a focus on the role and nature of 'awareness' in collaborative work.

Figure 2: UML Editor



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

1. Introduction

Computer Supported Collaborative Work (CSCW) has been defined [6] as follows:

CSCW [is] a generic term, which combines the understanding of the way people work in groups with the enabling technologies of computer networking, and associated hardware, software, services and techniques.

While there is much research going on on a broad range of issues related to CSCW, we will focus on a particular sub-area, namely Collaborative Software Engineering (CSE).

Software development in the industry today is a very complex task, requiring many developers to work together on different aspects of the product. Still, direct support for the collaboration required in these projects is absent from most tools used. Version control tools are the only ones that have some measure of collaboration built in, and frequently the only synchronization performed is the *nightly build*. Clearly, there is much scope for the development of new tools and methodologies, in order to introduce better support for collaboration amongst developers.

There is a wide range of applications that fall under the purview of CSCW. As illustrated in this figure (Fig.1), we can define a whole spectrum between two extremes, where on one extreme we have 'pessimistic' applications that have coarse-grained locking to prevent access to shared artifacts (such as CVS) and on the other extreme we have 'optimistic' applications that allow users independent access, and rely on merging for conflict resolution (such as interactive chat, or 'instant messengers'). As shown in the figure, ideal CSE applications should lie between these two extremes, with some combination of *fine-grained* locking and merging. For example, this figure (Fig.2) shows an example of a collaborative UML editing tool.

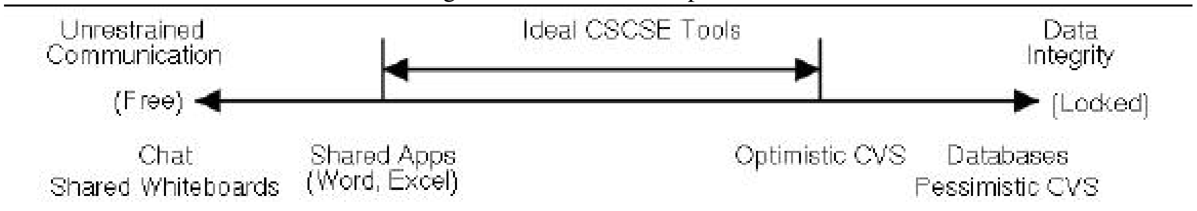
This survey looks at the theme of awareness in collaborative software engineering, and for this purpose it considers three different research areas. First, we consider augmenting CVS with a broadcast tool, and see how that affects developers using it. We take an in-depth look at a major collaboration tool, followed by a user study about its effectiveness. We then examine the role of interruption in a collaborative development environment, and how it affects productivity. Finally, we attempt to unify the results from these different areas of study with our central theme.

2. Augmenting CVS with notification

As mentioned previously, version control systems (e.g. CVS, Subversion) are the most widely prevalent form of collaboration in software engineering. These systems are very useful for co-ordination amongst developers working on a common project.

However, it has been observed that most of the actual communication is *informal*, and takes place via some other mechanism (e-mail or chat). Thus, it should be possible to

Figure 1: CSCW Tools Spectrum



combine both these spheres of interaction to some degree, improving the overall experience for the developer.

A recent user study [5] looks at the result of doing something similar – augmenting CVS with a chat/notification facility, thus allowing *mediated* communication, where the developer is more aware of what is going on.

2.1 Implementation

The authors noticed that CVS logs were used as an auxiliary form of communication, as developers referred to them in external (e-mail/chat) communication. They added an event notification system, Elvin, to broadcast log messages to a common tickertape. The client interface that displayed these messages also supported a chat mechanism.

Elvin is a publish/subscribe system which broadcasts events to the user's terminal, displaying, for each recent CVS log entry, the relevant group, the developer with whom the change is associated, and the contents of the entry. The authors studied whether there was any changes both in the nature of CVS messages, and the nature of communication among developers.

2.2 Observations

The biggest change the authors saw was in the nature of interaction amongst the developers. Knowing that other developers could 'track' their work, people paid more attention to entering information along with their CVS commits, and in fact supplemented their entries with more contextual information.

2.2.1 Communication

This information not only works to supplement comments with extra details about each change, but also helps document the *'flow'* of the work, providing a form of commentary to the project, which allows a developer to get an overview of an area of the project, at a glance. Also, this awareness of others' work stimulated discussion, as the messages on the tickertape became objects of conversation. CVS events could thus trigger a chat session among developers, thus leading to a merging of synchronous (chat) and asynchronous (CVS commits) forms of communication. This in turn allows developers to better co-ordinate their actions amongst themselves.

2.2.2 Awareness

The 'commit' operation transfers code from what is essentially a private workspace of a developer, into the public codebase. This augmentation of the version control system however, extended that to be an indicator of 'presence'. This also allows developers to be aware of each other's availability, which, as we will see later, allows for more efficient management of interruptibility.

3. CAISE

We have seen how version control systems are widely used for coarse-grained archival of code. In this section we will look at a fine-grained CSE tool that allows fine-grained software development. There has been much recent work on similar tools, and perhaps the best known recent commercial product with these features is the Jazz toolkit for Eclipse, developed by IBM.

CAISE [4], discussed here, is however, more of an academic product, developed by a group of researchers at the University of Canterbury, New Zealand, and shares many features with Jazz, although its underlying structure is different.

3.1 Motivation

In a typical software development environment, the only way a developer can be aware of code history is through CVS. There is no explicit way to determine whether another developer is working concurrently on the same piece of code, or on a higly dependent piece of code.

Thus, we need a CSE tool that will provide this awareness to the developer, for which it will be essential for the tool to have a semantic understanding of the codebase. A typical way to avoid the aforementioned situation is to lock common files so as to avoid potential conflict - this is often not required, and causes unnecessary delays. A good CSE tool should thus also support varying levels of granularity in locking.

3.2 Implementation

CAISE has a centralized client-server architecture [1]. This allows it to use more resources than a per-desktop model, and sustain higher throughput to co-ordinate a large team of developers. At the same time, it is highly extensible, and has a 'plugin' model to support a variety of client tools and widgets.

As seen in the adjoining figure (Fig.3), the server is responsible for maintaining a semantic model of the software - it stores information such as the symbol table and references between symbols. It interacts with language-specific parsers and analyzers to maintain and update the semantic model, and generate 'change events'. It also supports custom feedback modules which can generate client-specific events based on the state of the semantic model (such as tracking changes by a particular developer). The server also keeps track of user communication events which are generated by chat applications interacting with CAISE.

Having a centralized server architecture means that client-side IDE's can be lightwieght, and can maintain a local model of the code, perhaps containing only the current set of modules being worked on. The client tool can receive events from the server, and update its local model accordingly.

3.3 Awareness

CAISE provides many features to extend the traditional software development environment. Since the server maintains a complete semantic model of the software project, it is aware of *code dependencies* and *developer relationships*.

This allows the developer to be aware of the extent of his/her changes, both in terms of the modules/functions in the code that will be affected, as well as other developers who might face merge conflicts.

3.3.1 Client tools and Visualization

An example of a client tool interoperating with CAISE is shown in this figure (Fig. 3.3)- this shows the Borland Together IDE. Region 'A' shows a *change graph*, which lets the developr keep a visual tab to real-time changes in the codebase. Region B shows a *user tree*, which allows the developer to know the 'location' of other developers in the code, and helps him/her to avoid potential conflicts. Region C shows a message panel, where the develops receives feedback events. As mentioned earlier, these are user-defined events generated by the server, based on the state of the semantic model, according to criteria chosen by the developer. Thus, the developer is presented with a dashboard of sorts along with the traditional IDE, which can be customized as per his/her wishes.

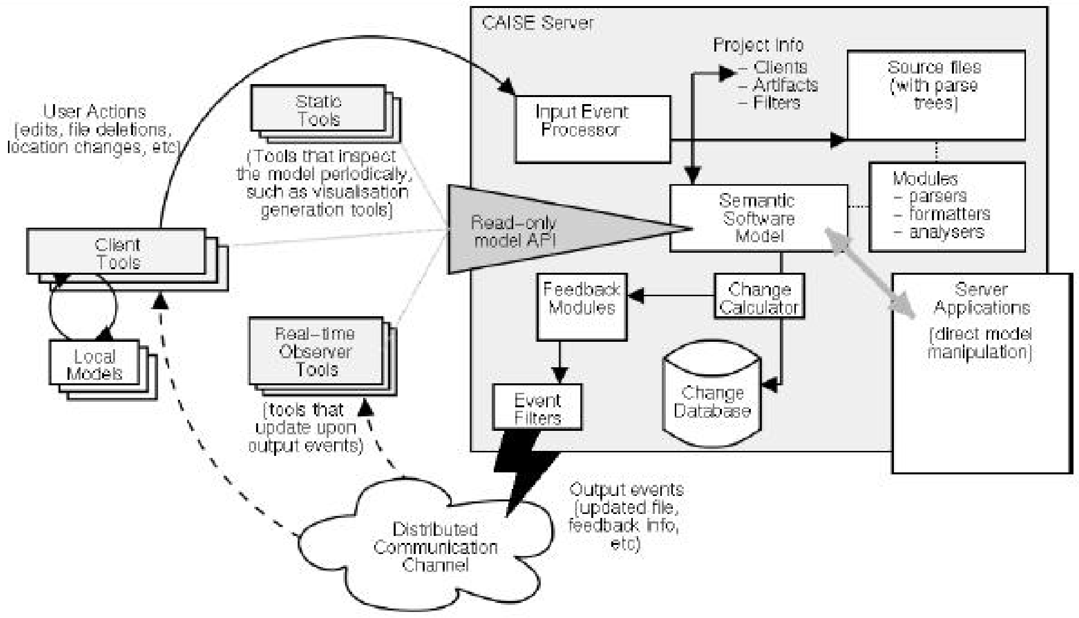
In addition to these examples, another useful category of client tools are visualizations of the event logs generated by the server. As CAISE supports XML logging, the team lead or project manager can access an enormous variety of information, such as events generated by a particular developer, or changes to the code base over time. This allows him to instantly get a snapshot of the project, and be aware of macro-level trends (such as growth of codebase/bug database by project team).

3.4 CAISE and CSE

CAISE provides all the *taskwork*-oriented features that traditional tools provide, and extends them to also support *teamwork*-oriented features, to indicate awareness. There are many modes that a developer might work in while using CAISE. Each mode reflects a different approach to conflict resolution, and a different level of awareness.

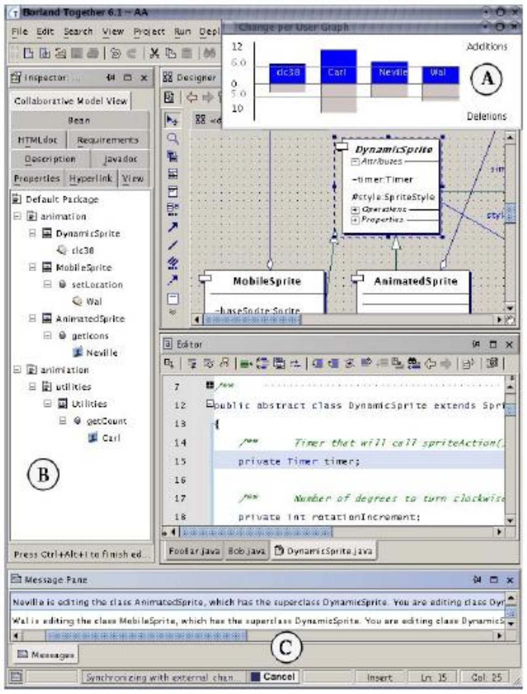
The traditional mode of working corresponds to the **private** mode, where a developer works on a chunk of code separately, and then re-integrates it into the common codebase.ls are visualizations of the event logs generated by the server. As CAISE supports XML logging, the team lead or project manager can access an enormous variety of information, such as events generated by a particular developer, or changes to the code base over time. This allows him to instantly get a snapshot of the project, and be aware of macro-level trends . However, now it is also possible to work in a real-time **independent** mode, where developers can work simultaneously on the code, while CAISE monitors the semantic relationship between the pieces of code they are working on, and alerts them to any potential conflicts. They can of course, choose to ignore these alerts and work in **melee** mode instead, which might be suitable for rapid changes to a confined region of code, where the developers communicate using some out-of-band

Figure 3: CAISE Architecture



CAISE Server
Project Info
– Clients
– Artifacts
– Filters
Source files (with parse trees)
Input Event Processor
User Actions (edits, file deletions, location changes, etc)
Static Tools
(Tools that inspect the model periodically, such as visualisation generation tools)
Read–only model API
Semantic Software Model
Modules
– parsers
– formatters
– analysers
Client Tools
Local Models
Feedback Modules
Change Calculator
Server Applications
(direct model manipulation)
Real–time Observer Tools
(tools that update upon output events)
Event Filters
Change Database
Output events (updated file, feedback info, etc)
Distributed Communication Channel

Figure 4: CAISE Client



```
/**
public abstract class DynamicSprite extends Spr
{
    /**      Timer that will call spriteAction(
    private Timer timer;

    /**      Number of degrees to turn clockwis
    private int rotationIncrement;
```

Neville is editing the class AnimatedSprite, which has the superclass DynamicSprite. You are editing class Dyn

Wal is editing the class MobileSprite, which has the superclass DynamicSprite. You are editing class DynamicS channel, such as audio, to synchronize their work and avoid conflict.

Additionally, due to the customized locking capabilities of CAISE, the team lead could opt to give the other developers a read-only view of a part of the code, following a WYSIWIS (what you see is what I see) model, where the other developers are essentially 'following the leader'. Finally, the customized user events supported by CAISE can allow developers to react to a predefined conflict situation, allowing the development team to have a comprehensive 'conflict policy'.

3.5 User Study

As mentioned previously, there are now many other tools that perform the same function as CAISE, so studying the benefits of CAISE is really representative of the benefits of CSE tools overall. A user study [2] was performed to test the benefits of CAISE to a team of developers.

3.5.1 Methodology

For the purposes of the study, the authors studied the behaviour of the users in two modes, the *conventional* single-user mode with version control, and the *collaborative* mode, with real-time sharing (in either WYSIWIS or *independent* mode).

They also made the users perform two kinds of tasks, each intended to test a different kind of collaboration benefits. *Intra-file* tasks, such as editing two pieces of code near each other in the same file, test merge conflicts. *Inter-file* tasks, such as editing two pieces of code in separate files, which share some relationship, test dependency conflicts.

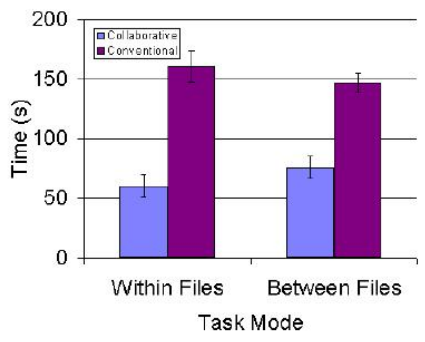
The mean task completion times were compared, for each mode and task.

3.5.2 Results

Not surprisingly, the authors found that in collaborative mode, the users performed better in collaborative mode. As seen in the adjoining figure (Fig. 3.5.2), the mean time for completion was nearly half, in collaborative mode, for both task categories, although intra-file tasks seem to have benefited more. This can probably be explained by the fact that knowing which developers are editing code in close proximity to your code avoids costly merge errors.

In addition to this study, they also asked the users to take a survey requiring them to answer questions about how CAISE was useful, requiring them to rate on a scale of

Figure 5: CAISE User Study Results



1 to 20. The users rated all points consistently between 14 and 16.

4. Interruptibility

Finally, we take a look at a (somewhat unrelated) study which focusses almost entirely, on the nature of *interruptions* in software teams. Developers working on a common project frequently interact in an informal setting. They take breaks from work, which may be social in nature, or with a functional purpose. They frequently interrupt each other either as part of the break or for communication purposes.

Interruptions thus affect teamwork to a large degree, and the authors of this [3] paper wanted to examine this aspect of collaborative software development in greater detail.

4.1 Methodology

To be able to better analyze the role of interruptions, the authors decided to observe two different teams of people. One team, referred to as "'Team Pair"', worked in an open area, in physical proximity to each other. The other, "'Team Solo"', worked alone, with e-mail, telephone, and a computer-mediated communication tool provided for collaboration. Over a period of seven months, the behaviour of developers on both teams was tracked, with emphasis on periods of work.

4.2 Observations

There were several interesting observations gleaned over the period of study. Fundamentally, Team Pair's interruptions and breaks were largely functional in nature, and short, while Team Solo's interruptions and breaks were social, and longer. This might be tied to a differentiating factor between both teams - the nature of distraction. Team Solo members, working remotely, were more easily distracted by people around them - and also, in some instances, by information overload from their communication tools.

Another aspect of communication the authors studied was the nature of interruptions. They concluded that it was easier for someone to interrupt a member of Team Solo because their 'interruptibility' was was harder to judge, as compared to Team Pair. Somone wanting to interrupt a developer on Team Pair is more *aware* of their state.

Additionally, interruptions (especially self-initiated ones) tended to happen when the developers were switching between multiple tasks, and Team Solo developers found it harder to keep track of multiple tasks. Developers on Team Pair 'preserved state' for each other, making it easier for one member of the team to pick up where he/she had left off, since the other member remembered where that was. This *awareness* of context allowed developers to handle interruptions better, and with more efficiency.

It seems that, to be productive, it is essential for CSE tools to help keep the developers aware, not only about their code, but also about multiple tasks, and to actively maintain context to allow the developer to manage interruptions better.

5 Awareness and Concurrency

We saw three seemingly disparate papers, on very different topics. However, all of them dealt with the issue of collaboration in the software engineering process, and going back to each of them, we can pick out several common factors.

We saw how augmenting a commonly used version control system (CVS) with a simple tool (notification) transformed developer relationships and productivity. In this case therefore, increased awareness lead to greater accountability and to a greater amount of contextual information being shared amongst developers.

CAISE was an example of a system that allowed developers to be aware of code dependencies and developer relationships in the code, thus allowing for increased concurrency in software development by avoiding potential conflicts. Awareness is thus related to conflict-resolution, although they are not necessarily orthogonal – we saw that for some situations (for example, the *melee* mode in CAISE) it might be optimal for developers to resolve conflicts through an out-of-band channel.

Finally, we saw how awareness is related to interruptibility in software teams, and how it affects teams of developers in different situations.

There are many ways that awareness impacts software development - it is certainly a key factor in collaboration, and tools built around the concept of providing the user with more awareness seem to have a positive impact on productivity. It is necessary to have a complete understanding of how to incorporate greater awareness in every part of the software development process, so that CSCW tools may be extended to allow for increased collaboration.

References

[1] Cook, C., Churcher, N., "Modelling and Measuring Collaborative Software Engineering", *28th Australasian Computer Science Conference*, 2005

[2] Cook, C., Irvin, W., Churcher, N., "A User Evaluation of Synchronous Collaborative Software Engineering Tools", *12th Asia-Pacific Software Engineering Conference (APSEC '05)*

[3] Chong, J., Siino, R., "Interruptions on Software Teams: A Comparison of Paired and Solo Programmers", *CSCW 2006*

[4] Cook, C., Churcher, N., Irwin, W., "Towards Synchronous Collaborative Software Engineering", *11th Asia-Pacific Software Engineering Conference (APSEC '04)*

[5] FitzPatrick, G., Marshall, P., Phillips, A., "CVS Integration with Notification and Chat: Lightweight Software Team Collaboration", *CSCW 2006*

[6] Wilson, P., "Computer supported cooperative work : an introduction.", *1991*