Software engineering

Software engineering has long been a potential field for researching collaboration. With technological development and economic means of increasing production, collaborative software engineering became a global phenomenon. This research is closely related to collaboration technology for collaborative software engineering: collaborative applications and infrastructures. Collaborative applications and infrastructures give better computer support for cooperative work than what has been offered by single-user applications.

Support for distributed collaborative software engineering helps to support cooperative work at all stages of collaborative software engineering: design, inspection, programming, debugging, and testing. Collaborative software applications offer many advantages, for example, potential savings in costs. Researchers are examining how computer supports a variety of collaborative processes including team decision making, writing, and budget planning.

It is of great significance because collaborative applications reduce team interaction costs in distributed collaborative software engineering: requirements analysis, design, inspection, programming, checkout, and testing. Infrastructure and Applications for Collaborative Software Engineering Prasun Dewan and Rajiv Choudhary (1995) investigated support for distributed collaborative software engineering. Their research findings are both important and promising. Research is showing promise of future success because collaborative applications have the potential for reducing software engineering costs.

For example, an application allowing distributed users to collect software requirements both synchronously and asynchronously could make travel cost smaller, as they could communicate with each other from different locations; reduce time spent in conferences, as they could complete more tasks asynchronously; and cut down documentation and maintenance costs, as the tool could automatically keep logs of events and the rationale behind design decisions (Diaper and Stanton 2004, p. 68).

In much the same way, a program editor can allow programmers to work more simultaneously; implicit locking-unlocking can reduce the overhead of checking-in-checking-out data; detailed access control can reduce the probability of programmers not responsible for some program component incorrectly modifying the component; and collaborative undo-redo can lower the effort that is needed to recover from team incorrect activities and allow new participants to playback the history. Dewan’ (1993) findings have indicated that some of these benefits do, actually, occur in such phases of collaborative software engineering such as design.

An extension of the abstraction of an editable active variable provides programmers with ability to control processes automatically and affords flexibility for implementing collaborative applications. Researchers have worked out the structure of several different kinds of software abstractions (and associated architectures) for supporting collaborative applications. These include abstractions that provide users with distributed sharing of a message bus, screen bitmaps, windows, widgets, and text buffers (Dewan and Choudhary 1995, p. 34).

The implementations of collaborative applications are requiring effort because, besides single-user interaction tasks, these applications must carry out collaboration tasks such as: dynamically making and breaking connections with (possibly distantly connected) users, multiplexing input from and demultiplexing output to multiple users, connecting the input-output of the users, providing executive system concurrency and access regulation, and offering joint undo-redo (Dewan 1993, p. 102). Usually these collaborative tasks may take three major forms.

First, programmers understand the task itself, that is, the goal to be achieved, the time available, the series of actions that must be followed. Second, programmers need a shared understanding about the marked limits of the cooperation; that is, they need to know what is part of the cooperative work required to be done, and what goes beyond the common workspace. Third, mutual understanding may be an objective itself. Programmers cooperate in order to discover and share some understanding in their task.

The functionality of a collaborative application is subdivided into the following functions: (1) Session Management allows programmers to start, end, connect to, and leave sessions with collaborative applications. (2) Collaborative interaction presents the effect of users' commands on their displays. (3) Coupled system determines which of the edits made by a programmer are shared with another programmer and when they are shared. (4) Remote Undo determines the effect of programmer undos on the displays of other programmers.

User Awareness allows programmers be aware of the steps taken by other programmers, (5) Access Control prevents unauthorized programmer commands. (6) Concurrency Control prevents incompatible concurrent programmer commands. (7) Differencing points out the differences between two different display states created by anisochronous coupled system between programmers. (8) Merging merges two different display states into a condition of single state (Defranco-Tommarello and Deek 2005, p. 5). These collaborative functions are different from single-user activities because they are invoked only when multiple programmers are using the application.

These are the functions that a collaboration infrastructure-groupware application generator must support. This dimensionalization of the design space of collaborative applications applies oneself to collaboration at the lowest, “ common object perception” level in alone and Crowston's collaboration hierarchy of coordination, group decision making, communication, and common object perception (Ocker 2001, p. 78). Examine, for example, a technology that was produced to support collaboration: a shared window system.

A shared window system became a development of a single-user window system that provides programmers with the same programming interface as the latter. It provides multiple programmers with opportunity to share physical copies of a logical window created by a client. It also at most times supports some form of floor control that makes certain that, at any time, only one programmer can provide input to the client. A client window can be

regarded as an “ active in-core database” because it is stored in-core and the system automatically keeps its physical copies coherent.

In this manner, this approach can provide programmers with the performance needed for supporting real-time system collaboration. In addition, because the programming interfaces of shared and single-user window systems are identical, the client is collaboration-transparent, “ that is, completely unaware it is being used by multiple, collaborating users” (Defranco-Tommarello and Deek 2005, p. 5). Considering the automation process, it is not having potential to do better, because a collaborative implementation is generated by the system without requiring the programmer to write any specification code.

The distributed system technology gives applications the necessary software flexibility and power. Separate processes can be created for different programmers, which can directly exchange ideas with each other to implement personal collaboration functions. In collaborative editing, a programmer interacts with an application by editing an active display of the data structures of the application. Different programmers edit different versions of the involved display and interconnection between the displays allows them to share their editing information without violating logicality and authorization constraints.

The cursor within the linkage circle acts as a user-application-controlled dial and shows that the combined functions are performed flexibly. Conclusion The area of collaborative applications is examining better computer support for cooperative software engineering than what has been provided by

traditional applications. Software engineering is a task characterized by sophisticated bending of the rules and conventions; hence, there is no algorithmic procedure for designing good software that enables computers to work together or productive cooperative tasks.

Instead, the engineering process is a continuous test for the original concept. It is possible that the final product has only indistinct similarity with the first outline. It is a process with high degrees of freedom. Therefore, better computer support for cooperative software can help programmers to save time, money, and effort in the course of prototyping useful applications. It is necessary to make further research on collaborative applications that cut down on team interaction costs in distributed collaborative software engineering.