Chapter 14
Voice-Based Approach for Surmounting Spatial and Temporal Separations

Kate O’Toole
GreenSun and Kastle Data Systems, USA

Srividhya Subramanian
University of Arizona, USA

Nathan Denny
University of Arizona, USA

ABSTRACT

This article describes a new voice-based tool for global collaboration. This tool, called EchoEdit, attempts to provide multimedia capabilities to program source code editing for the purpose of eliciting in situ vocal commentary from active developers.

INTRODUCTION

Global teams are becoming a growing reality. When teams are distributed, individuals lose the ability to walk down the hall and ask their other team members a question. When team members are distributed but work in similar time zones, they possess the ability to pick up the phone and call their peers. When the teams are distributed both temporally and spatially, team members lose these forms of immediate feedback, and different methodologies are needed to surmount the time and space separations that many global teams experience.

One way to distribute a team is by breaking the project down into different modules (e.g., site A is responsible for module A while site B is responsible for module B). In this situation, the two teams need to agree on an interface, but further interaction is limited. Another way for
Voice-Based Approach for Surmounting Spatial and Temporal Separations

distributed teams to interact is in a master-slave type relationship; in this case, one team may do the design work and a second team may do the testing. In this scenario, one site is telling the other site what to do and has more authority. A third way for distributed teams to interact is in what is known as the “24-Hour Knowledge Factory” (24HrKF) (Denny, Mani, et al., in press).

One way to reduce the confusion that may be caused when multiple people share the same piece of code is to increase the documentation and explain how the decisions were made. This article focuses on the use of voice recognition software for this purpose.

EMBEDDED PROGRAM DOCUMENTATION

Most programming languages have a built-in feature that allow programmers to make notes on what the code is doing and how the code works. These are usually done as comments and are denoted by a specific symbol that the compiler will ignore. However, many persons who have gone back to look at the code that they or their colleagues had written wish that code had been better documented and that they could access the logic that the original programmer used while creating that code. It is much easier to understand a code when someone tells you what he or she was doing.

Three surveys conducted in 1977, 1983, and 1984, highlight the lack of documentation as one of the biggest problems that the code maintenance people have to deal with (Dekleva, 1992). Prechelt conducted experiments using pattern comment lines (PCL) and added the pattern usage of design patterns to the code (Prechelt, Unger-Lamprecht, Philippson, & Tichy, 2002). By adding a few lines of comments and running experiments where students took existing code and modified it, major operational improvements were observed.

Documentation is not always a negative aspect of a project. When documentation is done correctly, it can be a positive aspect of the project and very useful. An example of good documenting on legacy software is IBM’s involvement with the space program. Starting in the 1970s, IBM worked with NASA to come up with programming for the space shuttle. The documenting process was high on the team’s to-do list when developing the software and finding and correcting bugs. Billings et al. cite that data collection to update and fix any errors paid off quickly with the software becoming nearly error free (Billings, Clifton, Kolkhorst, Lee, & Wingert, 1994). However, she further states that when this data was brought in late, it tended to disrupt the development activities. Timely documenting is as important as good documenting.

The oldest, and probably the most famous, code documentation approach is called literate programming. It was developed by Knuth in the early 1980s (Knuth, 1984). This style combines documentation with the code and then compiles the two into separate documents. There is a tangle function, which compiles the actual code into a program, and then the weave function, which produces documentation from the comments imbedded in the code. This style requires that the programmer have a good understanding of the weave function so that the comments come out properly formatted. The comments are written in the code document in a format that is similar to LaTeX.

Literate programming ideas have evolved into what is now called elucidative programming (Vestdam, 2003). In elucidative programming, the code and the documentation exist as different documents. Links are then added to the code so that the external documentation is easily accessed. There are no rigid standards for what the comments have to say, only that they should describe the functionality of the code.

Javadoc comments are another standard for documenting code (Kramer, 1999). Javadoc com-