Redwood: A Visual Environment for Software Design and Implementation

BRIAN T. WESTPHAL, FREDERICK C. HARRIS Jr., SERGIU M. DASCALU
Department of Computer Science
University of Nevada, Reno
Reno, NV 89557
USA
westphal@cs.unr.edu http://www.cs.unr.edu/~westphal/

Abstract: - This paper presents the main characteristics of Redwood, an integrated software development environment that proposes a novel solution for supporting software design and implementation activities via extensive use of predefined graphical templates (snippets), direct manipulation of programming constructs, and enhanced visual representation of program structure. The main design principles used in building the Redwood environment as well as the environment's overall organization and principal components are described. The current status of the Redwood project and several planned directions of future work are also presented in the paper.

Key-Words: - visual environment, software development, design, implementation, direct manipulation, snippets, graphical user interface, open source code, design tree, Redwood.

1 Introduction

The direct writing and manipulation of computer programming language source code is inefficient when dealing with a large number of abstractions. Most programs could be built by combining pieces of code that have previously been used successfully. However, the bulk of computer programming still involves rewriting again and again such pieces of code.

Through the use of open source software, part of this problem can be avoided. For example, if a new web browser package is to be developed, through open source software the developers can have access to a significant amount of code already written. In this particular example a package such as kHTML [1], developed for KDE [2] and used by systems such as Apple's Safari browser [3], can be used as a foundation. Thus, the developers can avoid rewriting large pieces of code and focus instead on the innovative aspects of their software.

Still, with open source software, developers are required to know a lot about the software they need to use and customize. Open source code is widely available over the Internet but, inconveniently, it is also widely spread over numerous sites. In practice, there are often several competing flavors of the same package. Moreover, development using an open source foundation is typically complicated by the lack of proper documentation. These problems make using open source packages impractical in many cases. For corporate use, open source software is insufficiently supported and not stable enough. For novice programmers or individuals that work in other fields (such as chemistry or physics) and resort to coding to increase their productivity, dealing with undocumented open source packages or even simply finding that a particular package exists requires significant effort. Such programmers do not have the knowledge of general resources and often do not know where to look to find software packages that can be trusted. To complicate things further, even when suitable software is available, it is typicallly undocumented to the point of incomprehensibility, except by expert programmers [4].

The Redwood environment, developed at the University of Nevada, Reno, USA, attempts to solve some of the previously mentioned problems. It also aims to handle several specific design and implementation issues. Redwood was primarily inspired by the visual programming systems Alice, created at Carnegie Melon University by the Stage3 research group [5, 6]. By making use of a graphical, drag-and-drop interface, a programmer can largely avoid implementation solutions that are platform and/or language dependent. Unlike Alice, however, Redwood strives to be a more complete solution for general programmers. Alice took the idea of drag-and-drop programming and geared it towards education, towards teaching programming and computer-based problem solving to students. Redwood is not specifically designed for students, it is designed for the wider community of programmers, including novice and occassional programmers, intermediate programmers, and experienced professional programmers.

Because the main concept on which the Redwood solution for software development support was founded is that of a *design tree* (an arborescent structure for abstract representation of program organization which is suitable for describing both functional and object-oriented code) we decided to give the name of a tree to this software development environment that supports both design and implementation. By envisioning it as open source code, we hope that the Redwood environment will grow to become as strong and reliable as its "natural counterpart" from which it took its name.

When the design process for Redwood began in Spring 2003, several goals were identified for the environment, namely: support for design hierarchy (hence, reliance on design tree structures), capability of direct manipulation via drag-and-drop, algorithmic independence, support for object-oriented design, support for parallel programming, open/shared source code, and support for documentation. In addition, it has been considered very important that the environment should be easy to understand, learn, and use. Later in the paper a concise evaluation of the degree to which these development goals have been achieved is included.

The remainder of this paper is organized as follows: *Section 2* presents the main design concepts and principles used in the construction of Redwood, *Section 3* provides details of the environment's interface and functionality, *Section 4* points to a number of related research and development areas that we intend to explore further, and *Section 5* concludes the paper with a summary of the environment's main characteristics.

2 Design Concepts and Principles

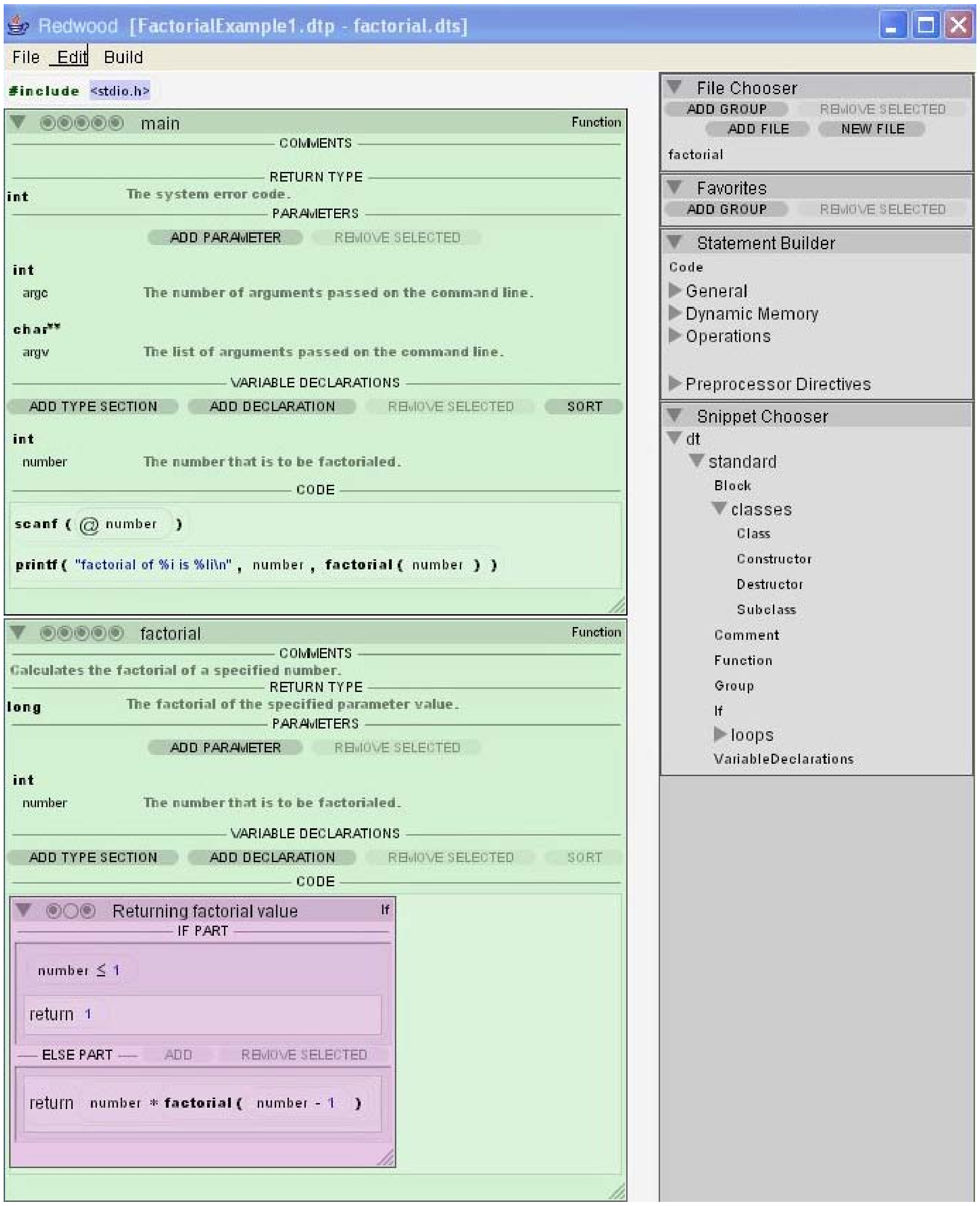
A major abstraction related to Redwood's visual design space is the tree structure that can be used to describe any program. Any program can be broken down into a tree-like structure and viewed at a "big-picture scale" (or high-level of abstraction, where only the program's major components are shown), at a "small-picture scale" (or low level of abstraction, where some specific details of the program's structure are visibile), or somewhere in between, at an intermediate level of abstraction. The direct tree manipulation mechanism present in Redwood makes viewing and editing complex code significantly more manageable. Through its graphical support for representing and manipulating design and programming constructs, Redwood, whose main window is shown in Figure 1, is meant to offer its users the known advantages of visual environments [7, 8], making it faster for developers to complete their tasks.

As a software engineering tool, Redwood can be used for fast corporate development as both the design and implementation aspects of a program can be built in a single space. In fact, the tree structure can be used to represent only the basic functions of a program, without implementing any code (that is, it can be used for design only). After the design tasks are completed, the programmers can fill in the implementation details in the tree. When design changes are needed, the design and the implementation (the details of the code) can be modified in a single place, thus eliminating the need to revisualize the design or the implementation spaces characteristic to more traditional types of software development environments.

The design principles employed in Redwood's construction allow a programmer to move easily between the large-scale representation of a problem's solution and its smaller details. Redwood does this by relying on several mechanisms and techniques, including templating via *snippets*, disclosure triangles and dots, and drag-and-drop manipulation. These are described next.

2.1 Snippets

In the Merriam-Webster Dictionary a snippet is defined as "a small part, piece, or thing". In Redwood, a snippet is used in the sense of a "a small part" of a solution. More precisely, a *snippet* is a template that describes a solution to a small part of a problem. The Redwood system ships with several types of snippets, most of which would be considered at the core part of a typical programming language. For example, *function* is a type of snippet that generally represents a typical computer program's function with a return type, parameters, and code. Redwood also comes with other snippets for *classes, loops, if-then-else* structures, and *blocks,*



```
#include <stdio.h>

main
int      The system error code.
int
  argc   The number of arguments passed on the command line.
char**
  argv   The list of arguments passed on the command line.
int
  number The number that is to be factorialed.

scanf ( @ number )
printf ( "factorial of %i is %li\n" , number , factorial ( number ) )

factorial
Calculates the factorial of a specified number.
long     The factorial of the specified parameter value.
int
  number The number that is to be factorialed.

Returning factorial value
number ≤ 1
return 1
return number * factorial ( number - 1 )
```

Fig 1. Redwood's Main Window

among other things. Each of these concepts represent a smaller part of the solution in a computer program.

In the typical sense, a function is defined by a set of rules according to a language grammar. These vary slightly from language to language, but in general a function has several key elements including a name, return type, parameters, and code. Virtually any language can be made, in one way or another, to duplicate the effects of a function. In the same way, while C++ has direct support for classes, virtually any other language can be made to synthesize class support.

A snippet in Redwood allows a programmer to describe an idea using functions, loops, classes, and other structures in a language-independent way – a very powerful feature that confers to the environment significant flexibility and scope of applicability.

A snippet is defined by its parts (as a function is defined by its name, return type, etc.) and its mappings to various languages. Figure 2 shows an example of XML code used to define a snippet.

```
<?xml version="1.0"?>
<!DOCTYPE SNIPPET>
<DEFINE-SNIPPET NAME="Function" BGCOLOR="210, 240, 210"
TITLE="name">
    <COMPONENTS>
        <COMPONENT TYPE="CommentEditor" NAME="COMMENTS"
    TITLE="COMMENTS"></COMPONENT>
        <COMPONENT TYPE="TypeEditor" NAME="RETURNTYPE"
    TITLE="RETURN TYPE" HASCOMMENT="TRUE"></COMPONENT>
        <COMPONENT TYPE="ParameterListEditor"
    NAME="PARAMETERS"
    TITLE="PARAMETERS"></COMPONENT>
        <COMPONENT TYPE="VariableDeclarationEditor"
    NAME="DECLARATIONS"
    TITLE="VARIABLE DECLARATIONS"></COMPONENT>
        <COMPONENT TYPE="CodeEditor" NAME="CODE"
    TITLE="CODE"></COMPONENT>
    </COMPONENTS>
    <PROTOTYPE LANGUAGES="C++">
$RETURNTYPE $FUNCTION.titleForIdentifier ($PARAMETERS);
    </PROTOTYPE>
    <TEMPLATE LANGUAGES="C++">
/**
$COMMENTS.formatForBlockComment
 *
$PARAMETERS.formatForBlockComment
 *
$RETURNTYPE.formatForBlockComment
 */
$RETURNTYPE $FUNCTION.titleForIdentifier ($PARAMETERS)
{
    $DECLARATIONS

    $CODE
}
    </TEMPLATE>
</DEFINE-SNIPPET>
```

Fig 2. XML Description of the Function Snippet with Support for C++ Output.

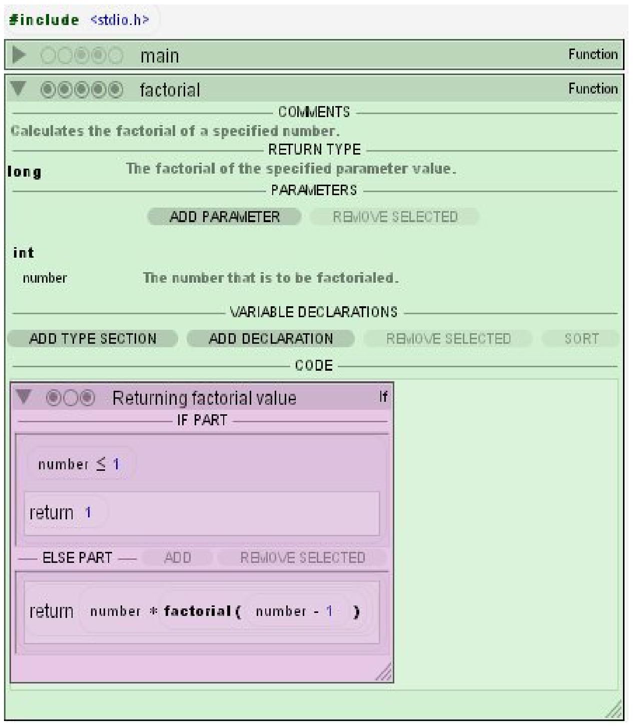
Due to their flexibility, snippets provide to developers a wide range of tools, including editors for code, data types, expressions, parameters, comments, and declarations.

Furthermore, snippets can be used to represent significantly more complex concepts than simple programming language constructs. For example, a snippet can be written to describe an algorithm such as one used for sorting. By encapsulating an algorithm into an object it is then possible to manipulate it in ways that are unavailable in traditional editing environments. With a sorting algorithm, for instance, a programmer may initially write code for a simple, easy to implement solution. Then, because of speed and efficiency requirements, it may be beneficial to replace the initial solution with a more complex, improved algorithm. Snippets, in such a case, allow the user to move the "algorithm object" around as if it were a physical object, dragging-and-dropping the new solution in to replace the initial algorithm. Further details on snippets creation and manipulation are given later in this paper.

2.2 Disclosure Triangles and Dots

After taking time to properly write and comment a piece of code, a developer may want to see only the comment attached to this piece of code. During the construction of the program, such a feature allows a developer to abstract away details of certain portions of code. Thus, with such an option, a programmer could better see the "big picture" of his or her solution, stepping away from minute details of the code and focusing on the overall logic of program. Redwood handles this concept through the use of *disclosure triangles* (a tool popular in many Apple software applications) for all snippets. The use of disclosure triangles at the snippet level allows programmers to reduce code into varying levels of complexity and depth by a simple click of a button.

In addition to disclosure triangles, a new visualization mechanism is used in Redwood to further customize the representation of a snippet. This mechanism that we have designed, denoted *disclosure dots*, allows a user to disclose or hide each individual part of a snippet. For example, a programmer may choose to see only the comments and code parts of a function, hiding the parameters and variable declarations parts of that function. Figures 3 and 4 demonstrate the use of disclosure triangles and dots.



```
#include <stdio.h>
```

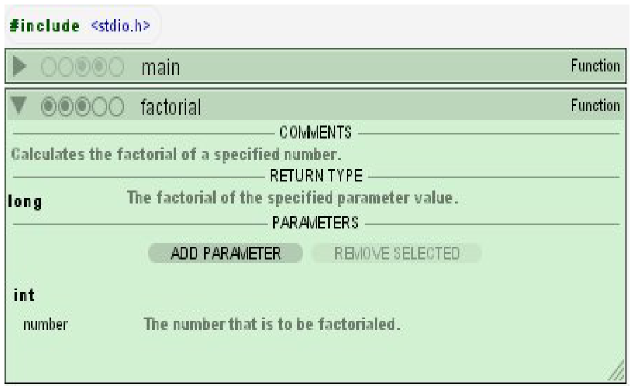
Calculates the factorial of a specified number.

long — The factorial of the specified parameter value.

int number — The number that is to be factorialed.

```
number ≤ 1
return 1
return number * factorial ( number - 1 )
```

Fig 3. Using Disclosure Triangles and Dots: All Compartments Shown in Function Factorial



```
#include <stdio.h>
main
factorial
COMMENTS
Calculates the factorial of a specified number.
RETURN TYPE
long        The factorial of the specified parameter value.
PARAMETERS
int
number      The number that is to be factorialed.
```

Fig 4. Using Disclosure Triangles and Dots: Two Compartments Hidden in Function Factorial

Specifically, the disclosure triangle can be seen in the upper left hand corner of each snippet. The disclosure dots are to the right of the disclosure triangles. The disclosure triangle of the `main` function is set to hide the contents of this function, whereas the disclosure triangle of the `factorial` function is set to show this function's contents. Further, the disclosure dots for the `factorial` function in Figure 3 are set to show the entire set of contents for the `factorial` function snippet, whereas the disclosure dots in Figure 4 are hiding the variable declarations and code sections of the same snippet. (Note that due to space limitations we have limited the complexity of the example to a simple `factorial` program.)

2.3 Drag-and-Drop Manipulation

One of the key elements to quickly manipulating large-scale objects is the ability to physically move them on the screen using the mouse. In contrast to traditional code editors, in Redwood programming constructs such as functions, classes, and algorithms can be manipulated on the screen an treated as individual "physical" objects. These objects can be dragged around on the screen to place, re-order, and replace components within the environment. In addition to being able to drag-and-drop snippets, programmers can also construct program statements and expressions almost entirely without the use of the keyboard. A programmer can drag a *function call* object, drag-and-drop the parameters for this function and then have a piece of code that can be easily "moved around" in the program design space.

This compelling ability of "dragging algorithms" not only increases the developers' productivity but also supports language-independent programming solutions, as verified code can be broken into individual pieces that can then be separately translated into other output languages.

3 Interface Details

As shown in the previously introduced Figure 1, the Redwood interface is composed of two major graphical areas. On the left, the *editing pane* provides the main space for manipulating programs. On the right, the *tools pane* is placed, supporting the actions perfomed in the editing pane by providing access to various drag-and-drop components.

3.1 Editing Pane

The editing pane of the Redwood environment takes up the majority of the screen real estate to show the structure and the details of the software being developed. The editing pane is built upon a nested tree structure in which each programming construct is placed in a snippet. Snippets can contain other snippets, and the level of complexity increases as one gets deeper into the tree. A programmer can flatten the depth of the tree and see only the desired level of detail by controlling the visual representation of the snippets via disclosure triangles and disclosure dots.

3.2 Tools Pane

The tool pane is currently made up of four *tool sub-panes* (several additional sub-panes, such as for accessing online libraries, will be included in future versions of the environment). These sub-panes allow programmers to select and manipulate multiple files within a project, handle favorite components (a kind of advanced clipboard for most commonly used items), and provide access to various draggable components, namely statement pieces and snippets.

The following subsections describe each of the current tool sub-panes in further detail, except for the file chooser sub-pane. The file chooser tool has been left out as its functions are similar to typical file manipulation tools found in traditional software packages.

3.2.1 Favorites

The *favorite tools sub-pane* can be used by the developer as an advanced clipboard. He or she can group favorite and repeatedly used items so that he or she can have a quicker, more convenient access to such items. Both statement pieces and snippets can be dropped onto the favorites tools sub-pane.

3.2.2 Statement Builder

The *statement builder tools sub-pane* provides the programmer with the ability to drag-and-drop statement pieces such as identifiers, literals, and assignment operations as physical objects on the screen. These are typically dragged from the statement builder tool and dropped into either the editor pane or into the favorite tools sub-pane. Once a statement piece is dropped, it can then be dragged from that point, creating a duplicate of the object when dropped. A snapshot of the *statement builder sub-pane* is shown in Figure 5.

3.2.3 Snippet Chooser

The *snippet chooser tools sub-pane* organizes snippets into directories based on relationships between items. A snapshot of this tools sub-pane is also shown in Figure 5.

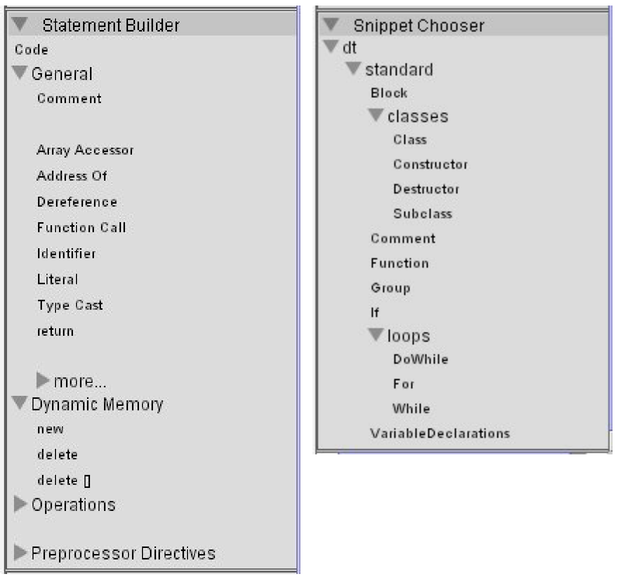


Fig 5. The Statement Builder and Snippet Chooser Tools Sub-Panes

4 Future Work

Currently available as a beta version, downloadable from [9], Redwood offers an opportunity for collaboration on open source projects. Also, within corporations large projects are often split between many programmers, and Redwood can efficiently help manage this type of internally shared code.

At this point in time, we assess that six out the eight major development goals for the environment mentioned in the Introduction section of this paper have been largely accomplished, namely: support for design hierarchy, capability of direct manipulation via drag-and-drop, support for object-oriented design, open/shared source code, familiarity of use (easy to understand, learn, and use), and support for commenting and documentation. Currently, we are working on the remaining two major objectives, support for parallel programming and algorithmic independence.

There are several directions in which Redwood can be improved. Perhaps the most urgent of these is concerned with the environment's usability. For example, it has been found that a pure drag and drop system is cumbersome to use when dealing with small tasks. The inclusion of shortcuts and/or automatic translation from typed syntax to Redwood structures could improve the tool's usability, especially for expert programmers who are skilled at rapidly typing source code into the computer.

Additional enhancements need also to be made to the editing system. For example, the use of cut, copy, and paste features is not yet available in the environment. Fortunately, the use of the *favorite* tools can reduce some of this need, but for increased convenience these tools will be added soon.

Another direction of further work relates to supporting additional languages and libraries. Currently, development support is available only for C and C++ programs, but we plan to extend in the near future this support to languages such Objective C/C++, Java, and Perl.

In addition to support for more programming languages, Redwood can also be equipped to formally handle pseudo-code level development as well as, to some extent, UML diagrammatic modeling and other software engineering notations and techniques [10].

Elements of other approaches for developing software and for building integrated development environments that we have suggested recently in other projects [11, 12], for example elements of stratified programming, could also be integrated to Redwood.

An improved version of Redwood could also incorporate hooks into several popular source code management systems such as the Concurrent Versioning System (CVS) [13], thus allowing sharing and managing of shared source documents. In addition, Redwood could help enforcing corporate documentation standards, optionally requiring, for example, that developers fully document their code before being able to check it into the CVS server.

Finally, Redwood could include support for an integrated online library of snippets. Even though the user community is expected to provide most of the support for extending the snippet library, a wide variety of powerful tools could also be included in the environment. Basic collections of frequently used data structures, sorting techniques, and input/output procedures are examples of packages that could be included in the environment.

These additions will make the Redwood environment a more complete design tool, giving developers a single platform on which they can productively build designs, prototypes, and implementations.

5 Conclusions

Redwood, a new visual environment for software design and implementation has been presented in this paper. The environment distinguishes itself through its advanced support for visual representation of program structure, direct manipulation of programming constructs via drag-and-drop, and extended support for program development based on the novel and powerful concept of snippets. In addition, the Redwood environment provides the foundation for a number of very interesting and challenging avenues of further research and development.

References:

[1] KDE HTML Widget, kHTML Library, accessed October 30, 2003 at http://devel-home.kde.org/~danimo/apidocs/khtml/html/index.html

[2] KDE Homepage, accessed October 30, 2003 at http://www.kde.org/

[3] Apple – Safari Website, accessed October 30, 2003 at http://www.apple.com/safari/

[4] The Ganssle Group, Open Source Quality?, accessed October 30, 2003 at http://www.ganssle.com/articles/opensrc.htm

[5] Alice: Free, Easy, Interactive 3D Graphics for the WWW, accessed March 10, 2003 at http://www.alice.org/

[6] Pausch, R., et al., A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality, *IEEE Computer Graphics and Applications*, May 1995.

[7] Green, T.R.G., and Petre, M., Usability Analysis of Visual Languages: a Cognitive Dimensions Framework. *Journal of Visual Languages and Computing*, Vol.7, 1996, pp. 131-174. Academic Press.

[8] Levialdi, S., Visual Languages: Concepts, Constructs, and Claims, *Proceedings of the 23rd International Conference on Information Technology Interfaces*, 2001, pp. 29-33. IEEE.

[9] Redwood v1.0 Beta 1 Homepage, accessed Oct. 30, 2003 at http://www.cs.unr.edu/redwood/

[10] Burnett, M.M., "Software Engineering for Visual Programming Languages", *Handbook for Software Engineering and Knowledge Engineering,* vol.2, pp. 77-92. World Scientific Publishing Co., 2001.

[11] Dascalu, S.M., Pasculescu, A., Woolever, J., Fritzinger, E., and Sharan, V., Stratified Programming Integrated Development Environment (SPIDER)", *Procs. of the 12th Intl.Conf. on Intelligent and Adaptive Systems and Software Engineering (IASSE-2003)*, July 2003, San Francisco, CA, USA, pp. 227-232.

[12] Westphal, B.T., Harris, F.C., Jr., and Fadali, S. Graphical Programming: A Vehicle for Teaching Computer Problem Solving, accepted at the *Frontiers in Education Conference (FIE '03)*, Boulder, CO, USA, November 2003.

[13] Concurrent Versions System, accessed October 30, 2003 at http://www.cvshome.org/