Design Aspects of the Redwood Programming Environment

Brian T. Westphal, Frederick C. Harris, Jr., Sergiu M. Dascalu
Department of Computer Science and Engineering
University of Nevada, Reno
Reno, NV, 89557 USA
{westphal, fredh, dascalus}@cse.unr.edu

Abstract *— Redwood is a development environment that supports drag-and-drop manipulation of programming constructs and visual representation of program structure. Redwood's architecture and functionality are based on the concept of snippet, defined loosely as a program component that encapsulates both a coding solution and its visual presentation. In addition, snippets support creation of unrestricted code libraries, thus fostering open-source development. This paper presents the motivation for Redwood, briefly overviews its functionality and mode of operation, and then focuses on the concepts underlying its design. Two essential parts of this design are the Snippet Display syntax (SDS) and the Snipplet Language (SL), created by the authors and presented in the paper. Implementation details, examples of use, and several directions of future work are also included in the paper.*

Index Terms *— Development environment, Redwood, snippet, visual programming.*

1 INTRODUCTION

The Redwood programming environment, whose main design elements are presented in this paper, is a project initiated in Spring 2003 at the University of Nevada, Reno. At that time, those involved in this project set forth a number of objectives for the environment, among them enhanced support for hierarchical program design, visualization and direct (via drag-and-drop) manipulation of programming constructs and components, algorithmic independence, inclusion of multiple programming languages, and open source software development [1, 2, 3].

An operational version, Redwood Beta 1, was made available in early 2004. Redwood Beta 1 demonstrated many of the key technologies necessary to implement a usable drag-and-drop programming environment. That release also led to formulations of new ideas and realizations that certain aspects have to be modified in order to create a truly functional development product. Beta 1 introduced the concepts of snippets, design trees, and disclosure dots [1]. Snippets provide a means by which generic programming constructs could be described. Design trees describe, in part, the relationships between the program components that make up a program. Lastly, disclosure dots, inspired from disclosure triangles in Mac OS X [4], work with snippets and design trees to allow visualization of source code at various levels of abstraction.

One of the most important lessons learned in developing the first release of Redwood, is that screen real estate is a precious resource for programmers [5, 6]. For Beta 2, the entire snippets engine had to be rethought and rewritten to better address this. Another lesson that became apparent while developing complicated programs using Redwood Beta 1 was that static templates are not sufficient for describing generic programming constructs. With the Beta 2 release, made publicly available in May 2005, one can create very powerful, dynamic templates.

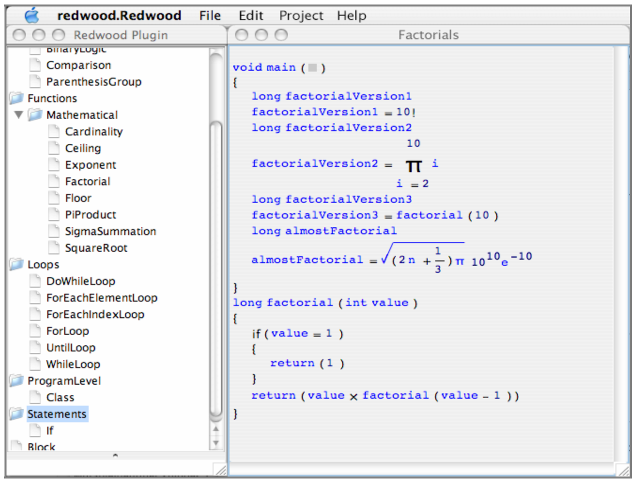
This paper presents recent results related to Redwood's latest version. Notably, although the authors remained truthful to the key concepts that define Redwood's development (snippets, design tree, disclosure dots, and drag-and-drop programming) they have radically re-designed the environment, which in terms of interface is currently refined to its most essential and elegant version to date. In terms of snippet manipulation and presentation, the new version is also significantly more powerful. The experience gained in undertaking Redwood's recent overhaul is reported here, through the presentation of the environment's new "look", details on the two "internal notations" created to build the latest version of Redwood (the Syntax Display Language and the Snipplet Language), and examples of use.

The remainder of this paper is organized as follows: Section 2 describes the motivation for creating Redwood and presents an overview of the environment, Section 3 outlines the principal architectural solutions used to build the environment, Section 4 provides details about the Snippet Display Syntax, Section 5 focuses on the Snipplet Language, Section 6 uses the sigma summation example to illustrate how programs are developed in Redwood, Section 7 briefly compares Redwood with related projects, and Section 8 concludes the paper with pointers to future work and a summary of the environment's significance.

2 Redwood: Rationale and Overview

From its very beginning, the Redwood programming environment (whose main browser is shown in Figure 1) has been intended as a tool that provides support for the large community of open source software developers. Another main idea behind Redwood's design has been to provide a user interface that is easy to learn, easy to use, quick to operate, and highly reliable. These two major objectives of Redwood's design have both been aimed at increasing the developers' efficiency as well as at enhancing their satisfaction when creating programs with this environment.

Redwood's distinguishing characteristics, particularly its support for graphical representation and direct manipulation of various program constructs and components, allows software creation in a visual workspace that is more intuitive and effective than that of a regular plain text code editor. In Redwood, developers can easily select program components (snippets) by clicking on their names in a tools panel (snippet chooser) and then drag-and-drop them to that place in the program's layout (program's structure) deemed to be the best for satisfying the program's requirements. Notably, the collection of snippets used for building programs in Redwood could grow constantly, as Redwood's library could be updated regularly via the Internet. Shown in Figure 1 are the snippet chooser plug-in panel (on the left-hand side of Redwood's main browser) and the editing panel (on the right-hand side of the browser).



```
void main ( )
{
   long factorialVersion1
   factorialVersion1 = 10!
   long factorialVersion2
                          10
   factorialVersion2 =    Π  i
                        i = 2
   long factorialVersion3
   factorialVersion3 = factorial (10 )
   long almostFactorial
   almostFactorial = √((2n + 1/3) π) 10^10 e^-10
}
long factorial (int value )
{
   if (value = 1 )
   {
      return (1 )
   }
   return (value × factorial (value - 1 ))
}
```

Fig. 1. Redwood's main browser

These are the core tools in Redwood for, respectively, selecting and organizing snippets. Note that the program shown in the editing panel of Figure 1 is included to illustrate the manipulation of various snippets. A more meaningful program in terms of algorithmic content is presented in Section 6 of the paper.

Writing and manipulating source code is often ineffective if it involves dealing with a large number of abstractions. With open source software (OSS) part of this problem is alleviated, as developers have access to a significant amount of code already written [7, 8, 9]. Therefore, they can avoid rewriting large portions of code and focus instead on the innovative aspects of their software. Unfortunately, software available as open source has its own drawbacks, for example it can be difficult to locate, poorly documented, not sufficiently supported, and not stable enough [8, 10, 11]. The Redwood environment attempts to solve some of these problems, as its architecture and functionality is intended not only for efficient program construction but also for effective manipulation of snippets, which are well-suited entities for supporting open source software development.

The snippets technology of Redwood, detailed later in the paper, is designed for promoting the reusability of code. Snippets give developers the power to encapsulate ideas, not just classes or functions, and to visualize program syntax in meaningful ways. For example, with a snippet one can use images or drawings to represent design concepts for which one might implement the code later. In addition, a future release of Redwood will be configurable so that software documentation can be enforced. These features of Redwood could help solve problems with open source development and programming in general. Developers do not have to rewrite time and again essentially the same code, as it could be made available through the environment's interface, both locally or remotely. In many software projects, customization of existing snippets could represent the only programming effort needed.

3 Internal Design Support

Snippets, the key elements of Redwood's design philosophy, can be described loosely as software components that encapsulate a coding solution (program logic) and its associated graphical representation (data for on-screen visualization). Snippets have been the subject of a previous publication, to which we refer the interested reader [2]. We mention only that a snippet, which can be as simple as an assignment statement or as complex as a very intricate algorithm, has an internal structure that consists of two parts: a display section and a template section.

A snippet's display and template sections are described using the specially created Snippet Display Syntax (SDS) and Snippet Language (SL, or simply Snipplet), respectively. The display section of a snippet defines the portions of the snippet that are visible to the user and the template section specifies the mapping from the visualized snippet to the programming language output (generated code). As detailed later in the paper, display representations are currently described using static XML while template sections are described using dynamic Snipplet scripts.

The SDS was designed to be a general-purpose interface description that is both easy to write and simple to parse. The SDS came about as part of a supporting technology designed by the authors for Redwood, the interface builder (now encapsulated as the com.bleugris.xml package [12]). The SDS is a convenient way to build interfaces without writing Java code, which can be tedious. This syntax allows one to place into the interface description any Java AWT or Swing components, including custom components, snippets, and snippet editors. The interface builder allows one to load interfaces dynamically from files.

The SL was devised for generating source code. As such, its design needed to revolve around parsing and string handling. In addition, scripts needed to have dynamic access to the elements of the design tree. The language was built primarily with two other scripting languages in mind, PERL [13] and JavaScript [14]. These languages are commonly used for generating HTML and JavaScript code for web pages. Still, they are not fully geared towards source code generation. In particular, code in both languages tends to become unstructured quickly, if not carefully managed. For source code generation, dealing with "languages inside of languages", this was something that had to be avoided. Like PERL and JavaScript, the SL makes use of variant types and has several built-in functions specifically designed for parsing. However, unlike PERL and JavaScript, SL's set is kept to a minimum, removing insecure features such as file I/O. The SDS and the SL are detailed, respectively, in the next two sections of the paper.

4 Snippet Display Syntax

A snippet is defined, using XML, in two sections. The first one, the display section, is immediately apparent to the user of the snippet (through visualization). The second one, the template section, is only noticeable when the user builds a program containing a snippet. This section discusses SDS, the syntax used for defining the display section of a snippet.

Within SDS, snippet display tags contain table-based layout of snippet editors. Using a table-based layout allows one to create non-linear or two-dimensional layouts. The structure of the table layouts is similar to HTML-style tables. That is, tables are organized into rows, and rows are organized into columns – they are "row major". Each cell can span one or more rows and columns and can be given width and height attributes. In addition, each cell can be given alignment parameters in both the horizontal and vertical directions.

Snippet editors are the foundation of snippet displays. Each snippet must contain at least one snippet editor. Snippet editors are special Java Swing components that extend the SnippetEditor class. Several snippet editors are included in Redwood. Each has various parameters that may be set. For example, the LineEditor snippet editor is commonly used to display a single line of text. The Editable parameter can be set to false so that the editor is used as a display only, and not as an input. The Text parameter is used to set the message of the editor. The complete description of SDL has been included in an internal report at [3] and is available upon request.

5 Snipplet Language

The Snipplet Language (SL, or just Snipplet) was created specifically to support source code generation. For inspiration, ideas from two languages commonly used for code generation, PERL and JavaScript, were used. The PERL or Practical Extraction and Report Language [13] is often used for CGI programming [15]. CGI allows one to dynamically create web pages, through generation of code such as HTML, JavaScript, and CSS. The JavaScript language, a derivative of Java, is often used to generate HTML and JavaScript code on the client-side [14]. Both languages have features and syntax that lend nicely to source code generation. In addition, several original ideas have been incorporated into the Snipplet language design.

Similar to JavaScript, variables in Snipplet are of variant type, meaning they can switch between types. The standard data types include integers, real numbers, strings, arrays (including multi-dimensional), and associative arrays (or hash tables). In addition to these, it is also possible to access snippets, giving scripts direct access to the design trees of programs.

While examining the features that a source code generation language should have, several significant factors come to mind. Most importantly, the syntax should be direct and simple, able to "stay out of the developer's way." Because one is working with two levels of syntax – the syntax for the language being used (Snipplet) and the syntax for the language being generated – it should be easy to distinguish between the two levels. At the same time, the language should be powerful enough and flexible enough to simplify the often-complex demands involved in source code generation.

Another important factor in designing a language for generating code is to provide a tendency for self-organizing syntax. That is, Snipplet code should not become confusing to examine due to lack of organizational formatting or overly compressed syntax. PERL programs often suffer from use of rather esoteric functions. Even though one can look up the meanings of various shortcut functions, inclusion of these in a language designed specifically for source code generation would be a mistake. Examples include the familiar "s///" function in PERL, which is a function for replacing substrings. Although compressed syntax can be convenient for the programmer, more straightforward naming conventions let programmers less experienced with the language or program interpret and modify code more easily. In fact, in Snipplet's case, even though the language is quite powerful, the grammar and the list of functions that a developer needs to know to work with it are relatively short.

The grammar for the Snipplet Language is shown, broken down into levels, in Figures 2 through 7. Level 0, shown in Figure 2, is the start symbol for the grammar. As one reaches higher levels in the grammar, productions are more specific.

```
1  template ::= block WS
```

Fig. 2. SL Grammar: Level 0, the start symbol

```
2  block ::= statement*
3  WS ::= ('\s+' | '//.*' | '/\*([\u0000-\u0029\u002B-
   \u9999]|\*[\u0000-\u002E\u0030-\u9999])*\*/')*
```

Fig. 3. SL Grammar: Level 1, basic template components

```
4  statement ::= WS (break | declaration | return |
   expression) WS ';' | WS (functionDeclaration | if |
   loop) WS
```

Fig. 4. SL Grammar: Level 2, statement types

```
5  break ::= 'break'
6  declaration ::= 'var' SP identifier (WS '=' WS
   (arrayInitializer | expression))?
7  expression ::= binaryArithmeticLevel5

8  //Sub-expressions
9  binaryArithmeticLevel5 ::= binaryArithmeticLevel4
   binaryArithmeticLevel5a?
10 binaryArithmeticLevel5a ::= WS '\|\|' WS
   binaryArithmeticLevel4 binaryArithmeticLevel5a?
11 binaryArithmeticLevel4 ::= binaryArithmeticLevel3
   binaryArithmeticLevel4a?
12 binaryArithmeticLevel4a ::= WS '&&' WS
   binaryArithmeticLevel3 binaryArithmeticLevel4a?
13 binaryArithmeticLevel3 ::= binaryArithmeticLevel2
   binaryArithmeticLevel3a?
14 binaryArithmeticLevel3a ::= WS '!=|<|<=|==|>=|>' WS
   binaryArithmeticLevel2 binaryArithmeticLevel3a?
15 binaryArithmeticLevel2 ::= binaryArithmeticLevel1
   binaryArithmeticLevel2a?
16 binaryArithmeticLevel2a ::= WS '\+|\-' WS
   binaryArithmeticLevel1 binaryArithmeticLevel2a?
17 binaryArithmeticLevel1 ::= binaryArithmeticLevel0
   binaryArithmeticLevel1a?
18 binaryArithmeticLevel1a ::= WS '\*|/|%' WS
   binaryArithmeticLevel0 binaryArithmeticLevel1a?
19 binaryArithmeticLevel0 ::= mainExpressionPart
20 //End sub-expressions

21 return ::= 'return' SP expression
22 functionDeclaration ::= 'sub' SP identifier WS '\{' WS
   block WS '\}'
23 if ::= ifPart (WS elseIfPart)* (WS elsePart)?
24 loop ::= dowhileLoop | foreachLoop | forLoop |
         whileLoop
```

Fig. 5. SL Grammar: Level 3, statement specifications

```
25 identifier ::= arrayIdentifier | scalarIdentifier |
   hashIdentifier
26 arrayInitializer ::= '\{' WS (expression (WS ',' WS
   expression)* WS)? '\}'
27 typeCast ::= '\(' WS ('string' | 'real' | 'integer') WS
   '\)'
28 mainExpressionPart ::= (typeCast WS)?  (assignment |
   doubleValue | functionCall | identifier | longValue |
   parentheses | string | unaryArithmetic |
   snippetEditorFunctionCall)
29 ifPart ::= 'if' WS condition WS '\{' WS block WS '\}'
30 elseIfPart ::= 'elseif' WS condition WS '\{' WS block
   WS '\}'
31 elsePart ::= 'else' WS '\{' WS block WS '\}'
32 dowhileLoop ::= 'do' WS '\{' WS block WS '\}' WS
   'while' WS condition
33 foreachLoop ::= 'foreach' SP identifier WS '\(' WS
   expression WS '\)' WS '\{' WS block WS '\}'
34 forLoop ::= 'for' WS '\(' (expression | declaration) WS
   ';' WS expression WS ';' WS expression WS '\)' WS '\{'
   WS block WS '\}'
35 SP ::= ('\s+' | '//.*' | '/\*([\u0000-\u0029\u002B-
   \u9999]|\*[\u0000-\u002E\u0030-\u9999])*\*/')+
36 whileLoop ::= 'while' WS condition WS '\{' WS block WS
   '\}'
```

Fig. 6. SL Grammar: Level 4, primary support for statements

6 WRITING PROGRAMS IN REDWOOD

To use an existing snippet, one may simply drag a snippet, listed by name in the snippet chooser tool, and drop it into

```
37 arrayIdentifier ::= scalarIdentifier ('\[' expression?
   '\]')+
38 assignment ::= identifier WS ('=' | '\+=' | '\-=' |
   '\*=' | '/=' | '%=') WS expression
39 doubleValue ::= '(\+|\-)?[0-9]+\.[0-9]+'
40 functionCall ::= identifier WS '\(' WS parameters? WS
   '\)'
41 hashIdentifier ::= scalarIdentifier '\{' expression?
        '\}'
42 longValue ::= '(\+|\-)?[0-9]+'
43 parentheses ::= '\(' expression '\)'
44 scalarIdentifier ::= '[\$@A-Za-z_][A-Za-z_0-9]*'
45 snippetEditorFunctionCall ::= '\[#' WS expression (WS
   ':' WS parameters)? WS '\]'
46 string ::= '"(\\"|[\u0000-\u0021\u0023-
   \u9999])*"|\'(\\\'|[\u0000-\u0026\u0028-\u9999])*\''
47 unaryArithmetic ::= '!' WS expression
48 condition ::= '\(' WS expression WS '\)'
```

Fig. 7. SL Grammar: Level 5, secondary support

the editing space. Once in place, a snippet can be repositioned and manipulated as needed. A newly dropped snippet is called a visualized snippet. These snippets are ready for customization. Not all snippets require customization; some may meet one's needs immediately upon being dropped. However, most snippets will need at least minor customizations. A snippet can be customized in two ways. With some types of snippet editors, editing text and/or manipulating controls will help customize the snippet. For other types of snippet editors, one may drop additional snippets. For example, in a CodeEditor one may drop as many snippets as necessary. In an ExpressionEditor only a single snippet may be dropped.

The Select Build Language option of the Project menu allows a programmer to select the language desired for output. The current Beta 2 release of Redwood supports C, C++, and Java output for all included snippets. The Project menu's Build option runs the Snipplet scripts for the project generating source code output in the desired language. This code is placed in source files as appropriate.

Figures 8 and 9 form the two portions of the Sigma Summation snippet. Figure 8 contains the display tag, which describes the visual elements of the snippet. Figure 9 contains the template tag. The template tag describes how source code is generated based on the customizations made to the snippet. Figure 10 demonstrates the use of the Sigma Summation snippet in Redwood. The code in the figure performs the naive matrix multiplication algorithm [16]. Because of the graphical environment, programming constructs can be described in their natural form. Figure 11 is the equivalent code written in Java. While it is relatively easy for a programmer to decipher either version, Figure 10 displays a more compact depiction.

7 RELATED WORK

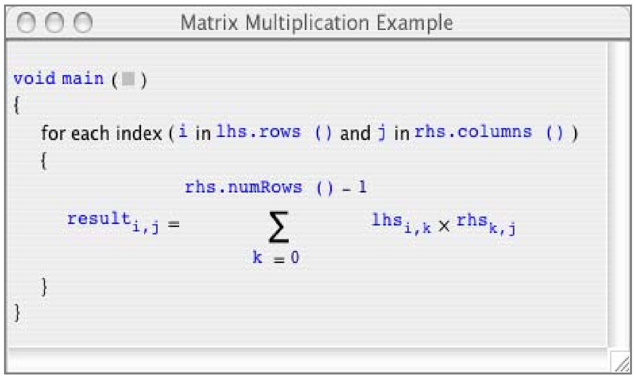
In its early stages, Redwood has been inspired primarily by Alice, a visual programming system developed at Carnegie Melon University by the Stage 3 Research Group [17, 18]. Alice proposed the idea of visual, drag-and-drop programming as a means for teaching computer-based problem solving and computer programming to high school and university students. However, Redwood is not designed specifically for use by students. Redwood's scope is larger, as it aims at providing a useful tool for the much larger open source software community, including beginners, intermediate, and experienced programmers. Of course, based on its main features (visual representation and direct manipulation), Redwood fits in the category of visual programming environments [19, 20].

```
<Snippet type="Math.Sigma Summation"
extends="Statement, Expression"><display><table>
 <tr>
  <td width="8"/>
  <td valign="center">
   <table>
    <tr><td halign="center"><ExpressionEditor
     name="haltingvalue">
     <param name="MinimumSize"><Dimension>
      <param name="Size"><int value="10"/><int
       value="10"/></param>
     </Dimension></param>
    </ExpressionEditor></td></tr>
    <tr><td halign="center"><LineEditor>
     <param name="Text"><String
      value="&#x2211;"/></param>
     <param name="FontSize"><int value="24"/></param>
     <param name="Editable"><boolean
      value="false"/></param>
    </LineEditor></td></tr>
    <tr><td halign="center"><table><tr>
     <td><IdentifierEditor name="loopvariable">
      <param name="Text"><String value="i"/></param>
      <param name="MinimumSize"><Dimension>
       <param name="Size"><int value="4"/><int
value="-
        1"/></param>
      </Dimension></param>
     </IdentifierEditor></td>
     <td><LineEditor>
      <param name="Text"><String value=" =
"/></param>
      <param name="Editable"><boolean
       value="false"/></param>
     </LineEditor></td>
     <td>
      <ExpressionEditor name="initialvalue">
       <param name="Snippet">
        <Snippet type="Numeric Value">
         <LineEditor name="value">
           <param name="Text"><String
            value="0"/></param>
         </LineEditor>
        </Snippet>
       </param>
       <param name="MinimumSize"><Dimension>
        <param name="Size"><int value="10"/><int
         value="10"/></param>
       </Dimension></param>
      </ExpressionEditor>
     </td>
    </tr></table></td></tr>
   </table>
  </td>
  <td valign="center">
   <ExpressionEditor name="expression">
    <param name="MinimumSize"><Dimension>
     <param name="Size"><int value="10"/><int
      value="10"/></param>
    </Dimension></param>
   </ExpressionEditor>
  </td>
  <td width="4"/>
 </tr>
</table></display>
```

Fig. 8. Snippet display of Sigma Summation

```
<template language="Java">
 sub calculateReturnValue
 {
  var uid = getUID ();

  var expressionReturnValue = [#"expression" :
   "+Expression:return calculateReturnValue ();"];

  var resultType = [#"expression" :
"+Expression:return
   determineResultType ();"];

  var result[] = {
   "
   " + resultType + " Snippet_SigmaSummation_sum_" +
   uid + " = new " + resultType + " ();
   for (RedwoodDouble " + [#"loopvariable"] + " = " +
    [#"initialvalue"] + ";
   " + [#"loopvariable"] + ".lessThanOrEqualTo (" +
   [#"haltingvalue"] + ").booleanValue (); " +
   [#"loopvariable"] + " = " + [#"loopvariable"] +
   ".add (new RedwoodDouble (1)))
    {
     " + expressionReturnValue[0] + "
     Snippet_SigmaSummation_sum_" + uid + " =
      Snippet_SigmaSummation_sum_" +
     uid + ".add (" + expressionReturnValue[1] + ");
    }
    ",
    "Snippet_SigmaSummation_sum_" + uid
   };
   return result;
 }

  var returnValue = calculateReturnValue ();
  return returnValue[0];
 </template></Snippet>
```

Fig. 9. Snippet template of Sigma Summation (Java output)


Matrix Multiplication Example

```
void main ( )
{
   for each index ( i in lhs.rows () and j in rhs.columns () )
   {
```
$$result_{i,j} = \sum_{k = 0}^{rhs.numRows\ () - 1} lhs_{i,k} \times rhs_{k,j}$$
```
   }
}
```

Fig. 10. Matrix Multiplication with Sigma Summation Snippet

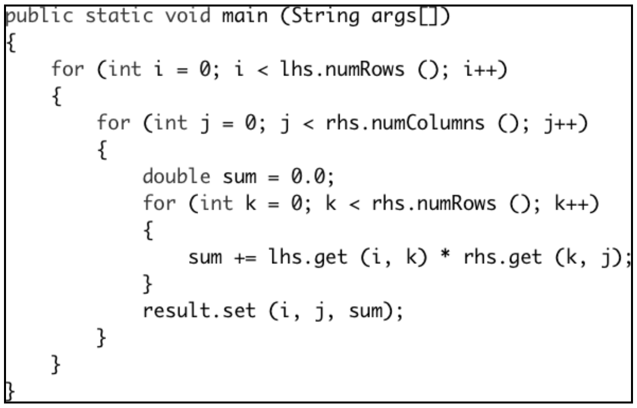

```
public static void main (String args[])
{
    for (int i = 0; i < lhs.numRows (); i++)
    {
        for (int j = 0; j < rhs.numColumns (); j++)
        {
            double sum = 0.0;
            for (int k = 0; k < rhs.numRows (); k++)
            {
                sum += lhs.get (i, k) * rhs.get (k, j);
            }
            result.set (i, j, sum);
        }
    }
}
```

Fig. 11. Matrix Multiplication in Java

Examples of environments that have similarities with Redwood include Prograph [21], LabView [22], CODE [23] and several others. Nevertheless Redwood is distinct from them in at least one of the following: execution model, application domain (e.g., LabView is best suited for engineering applications: test, measuring and control [24]), program components used (in this respect, snippets seems to be a concept quite unique to Redwood, at least in the sense we use it), and user interface ("look and feel"), which is clearly Redwood specific. This is not to say that Redwood is better than these languages, as for example the above environments supports parallel programming while Redwood does not at this point in time.

It is fair to say that, first, we are currently concerned with actually enhancing Redwood's present capabilities and plan for the near future a comparative study with other visual environments and, second, most of the above environments are specialized, and good or very good in some respects. We believe that Redwood is also good at what it does (or is characterized by) including support for general program development, "smooth" (that is, streamlined, simple and refined) user interface, support for multiple languages, extensibility, and flexibility.

8 FUTURE WORK AND CONCLUSIONS

In this paper the main design solutions used for creating Redwood were described. We believe that this novel programming environment, based on the key concept of snippets and employing the "drag-and-drop programming" paradigm, offers attractive, efficient and comprehensive support for software development.

Future releases of Redwood will include additional programming support and more usability features. Snippet displays will allow for dynamic content and the environment will be stress-tested for efficiency, reliability, and ease-of-use. Besides general improvements that can be made to the system, we have plans to include a large collection of pre-built snippets in the environments, in addition to the online snippet library. This set will include tools for parallel programming, mathematical representations of programming constructs, and templates for commonly-used structures such as loops.

One of the most useful features of Redwood is that it allows one to "extend" languages. With a plain text programming language, a developer is confined to standards sometimes defined decades before. The developer can add instances of new structures such as classes and functions, but he or she cannot create new types of structures. He or she cannot, for example, create a new type of loop in C++ (such as an until loop), which might be more direct (than a while loop) for solving certain problems. Redwood is about designing and programming software using natural techniques, where the environment and language do not get in the developer's way (on the contrary, it is intended to gracefully and efficiently support him or her). The system creates a supporting environment for a programmer to work and think effectively.

REFERENCES

[1] Westphal, B. T., Harris, F. C., Jr., and Dascalu, M. S. (2004) "Redwood: A Visual Environment for Software Design and Implementation", in WSEAS Trans. on Computing, 2(3):380-386.

[2] Westphal, B. T., Harris, F. C., Jr., and Dascalu, M. S. (2004) "Snippets: Support for Drag-and-Drop Programming in the Redwood Environment", in Journal of Universal Computer Science (JUCS), 2004, 10(7):859-871.

[3] Westphal, B. T. (2004) "The Redwood Programming Environment", Master's thesis, University of Nevada, Reno, NV, USA.

[4] "Mac OS X Panther Operating System", retrieved March 1, 2005 at http://www.apple.com/macosx/

[5] Burnett, I., Baker, M.J., Bohus, C., Carlson, P., Yang, S., and Van Zee, P. (1995) "Scaling Up Visual Programming Languages", IEEE Computer, vol. 28, no. 3, March, pp. 45-54.

[6] Na, L., Hosking, J, and Grundy, J. (2004) "Integrating a Zoomable User Interfaces Concept into a Visual Langauge Meta-Tool Environment", Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing, pp. 38-40.

[7] Wang, H. and Wang, C. (2001) "Open Source Software Adoption: A Status Report", IEEE Software, vol. 18, no.2, pp. 90-95.

[8] Lawton, G. (2002) "Open Source Security: Opportunity or Oxymoron?", IEEE Computer, vol. 35, no. 3, March, pp. 18-21.

[9] van Krogh, S. G., Spaeth, S., and Haefliger, S. (2005) "Knowledge Reuse in Open Source Software: An Exploratory Study of 15 Open Source Projects", Proceedings of the 38th Annual Hawaii International Conference on Systems Sciences (HICSS'05), pp. 1-10.

[10] Ganssle (2000) The Ganssle Group. "Open Source?", retrieved February 20, 2005 at http://www.ganssle.com/articles/opensrc.htm

[11] Fitzgerald, B. (2004) "A Critical Look at Open Source", IEEE Computer, vol. 37, no. 7, July, pp. 92-94.

[12] "com.bleugris.xml v1.07 Programmer's Manual", retrieved March 1, 2005 at http://www.bleugris.com/java/docs/com_bleugris_xml.html

[13] "Perl.com: Documentation", retrieved February 20, 2005 from http://www.perl.com/pub/q/documentation

[14] "Core JavaScript Reference 1.5", retrieved Feb. 2005 from http://devedge.netscape.com/library/manuals/2000/javascript/1.5/reference

[15] Kew, N. (2000) "CGI Programming FAQ", retrieved February 22, 2005 from http://www.htmlhelp.com/faq/cgifaq.html

[16] Foster, I. (1995) "Matrix Multiplication", retrieved February 28, 2005 from http://www-unix.mcs.anl.gov/dbpp/text/node45.html

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

[18] "Alice: Free, Easy, Interactive 3D Graphics for the WWW", retrieved February 28, 2005 from http://www.alice.org/

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

[20] Burnett, M.M. (2001) "Software Engineering for Visual Programming Languages", Handbook for Soft. Eng. and Knowledge Engineering, vol. 2, pp. 77-92. World Scientific Publishing.

[21] Cox, P.T., Glaser, H., and MacLean, S. (1998) "A Visual Development Ernvironment for Parallel Applications", Proceedings of the 1998 IEEE Symposium on Visual Languages, pp. 144-151.

[22] Wang, J.Z. (2003) "LabView in Engineering Laboratory Courses", Procs. of 33rd IEEE Frontiers in Education Conf., vol. 2, pp. FE13.

[23] Newton, P. and Browne, J.C. (1992) "The CODE 2.0 Graphical Parallel Programming Language", Proceedings of the ACM International Conference on Supercomputing, pp. 167-177.

[24] Josifovska, J. (2003) "The Father of LabView", IEE Review, Virtual Instrumentation, October, pp. 32-35.