University of Nevada, Reno

The Redwood Programming Environment

A thesis submitted in partial fulfillment of the

requirements for the degree of Master of Science

with a major in Computer Science.

by

Brian T. Westphal

Dr. Frederick C. Harris, Jr., Primary thesis advisor

Dr. Sergiu M. Dascalu, Secondary thesis advisor

August 2004

http://www.cs.unr.edu/redwood/

Abstract

Redwood is a visual environment for software design and implementation that allows programmers to manipulate code using several new methods proposed by the author. Through the use of two primary tools, snippets and design trees, programmers are able to view programs at varying levels of complexity and output code to multiple languages and highly variant machine architectures. The environment provides a single space in which one may both design and implement software solutions, thus potentially reducing overall time spent in the design and implementation processes. This document describes the motivation for Redwood, the concepts underlying its design, implementation details, results of use, and several directions of future work.

Acknowledgements

I would like to give special thanks to my thesis advisors, Dr. Frederick C. Harris and Dr. Sergiu M. Dascalu for all their support during Redwood's development process, for their help in editing this thesis, for their diligence on previous publications, and for their continued efforts and enthusiasm towards Redwood.

I would also like to thank my thesis committee member, Dr. M. Sami Fadali, for his invaluable comments and guidance in the editing process.

Table of Contents

List of Figures

List of Tables

Chapter 1 – Introduction

1.1 What is Redwood?

Developed recently in the Computer Science and Engineering Department at the University of Nevada, Reno, Redwood is an integrated development environment that allows programmers to create and visualize source code in a new way. The environment was designed from the ground up with problem solving support for open source developers in mind. Sharing code and collaborating in a visual workspace is much more intuitive and effective than when using plain text code editors. Redwood's drag-and-drop interface allows developers to select program constructs and components by simply clicking on entries in a tools panel. With Redwood's plans for a built-in online library of snippets, programmers may even be able to create new software with little or no code at all!

This project was initially inspired by Carnegie Melon's Alice software package [1, 2]. Although Alice is an excellent program for instruction, Redwood has been conceived with a much more powerful interface, primarily based on drag-and-drop manipulation of program components – the main idea being that the interface should not get in the programmer's way, but should instead be a tool for improving the programmer's efficiency. As the project progressed, Redwood has incorporated additional features that further distinguish it in the landscape of visual programming environments.

1.2 Project Goals

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 repeatedly rewriting such pieces of code.

With 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 [3], developed for KDE [4] and used by systems such as Apple's Safari browser [5], 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 typically undocumented to the point of incomprehensibility, except by expert programmers [6].

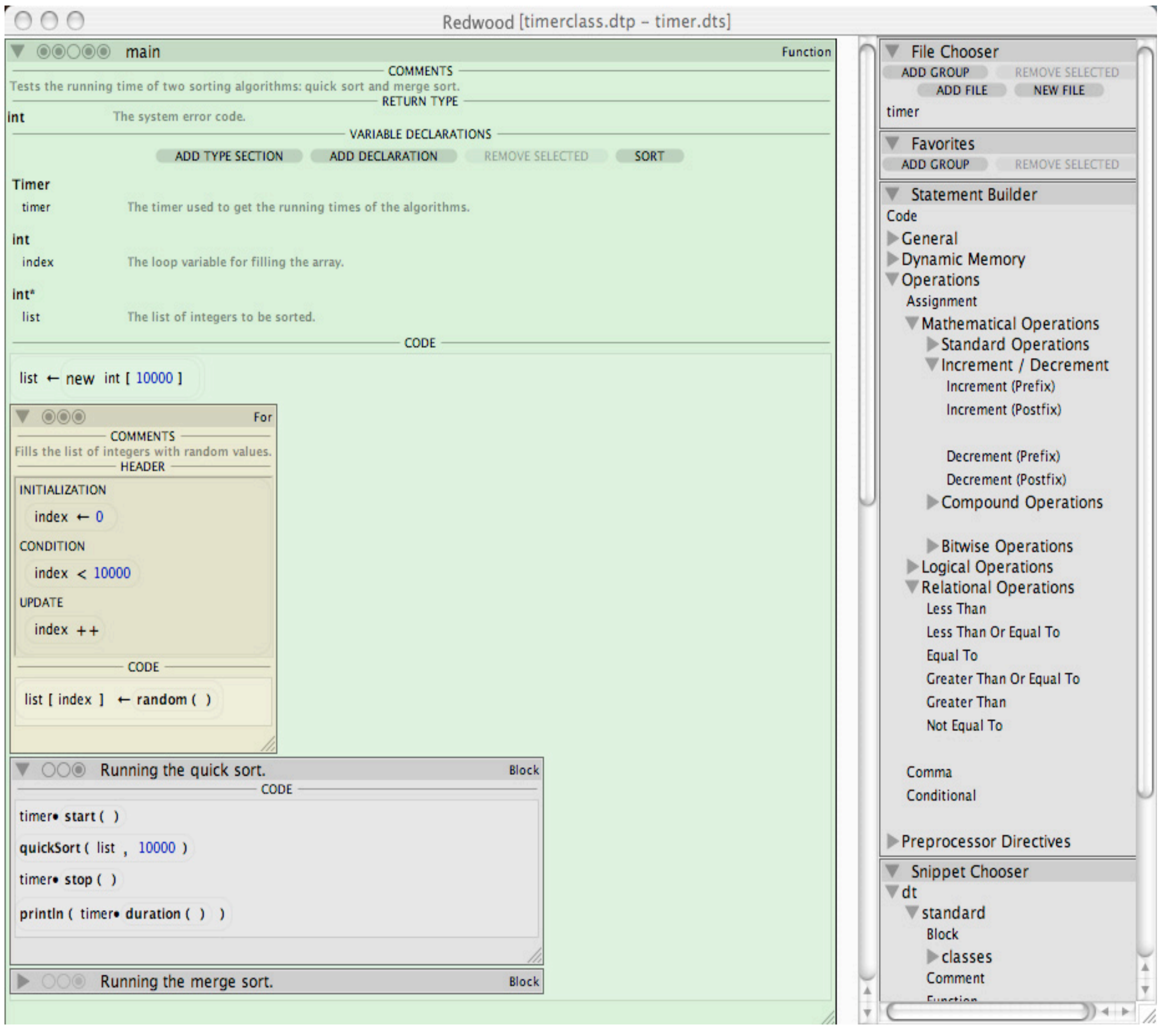
The *snippets technology* of Redwood is an excellent tool for promoting the reusability of code. Snippets give developers the power to encapsulate ideas, not just classes or functions, and to visualize syntax in a meaningful way. With a snippet one can, for instance, use images or drawings to represent design concepts for which one might implement the code later. A future release of Redwood will include an integrated, online library of snippets. In addition, a future release of Redwood will be configurable so that software documentation can be enforced. These features of Redwood will help solve problems with open source development and programming in general. Developers will not have to rewrite the same code; it will be available from the development interface. Only customization of existing snippets will be necessary for many projects.

1.3 Project Evolution

Redwood came about as part of an amalgamation of ideas put together into one project. Initially, in Spring 2003, a design goals document was completed, listing several key directions for developing a new IDE: enhanced support for hierarchical design, drag-and-drop, algorithmic independence, object-oriented programming, parallelism, usability, open source, and documentation. After the original design document was created, the environment underwent concentrated development that led to an initial operational version, Redwood Beta 1.

Redwood Beta 1 demonstrated many of the key technologies that would be necessary to implement a usable drag-and-drop programming environment. That release also led to formulations of new ideas and realizations that certain aspects would have to be modified to create a truly functional development product. Beta 1 introduced the concepts of *snippets*, *design trees*, and *disclosure dots* to programming. Snippets provide a means by which generic programming constructs could be described. Design trees describe, in part, the relationships between programming constructs. Lastly, disclosure dots, used in conjunction with disclosure triangles familiar to Mac OS X [7], work with snippets and design trees to allow programmers to visualize source code at varying 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. For Beta 2, the entire snippets engine had to be rethought and rewritten to better consider 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, one can create very powerful, dynamic templates. The complete Beta 2 release will be publicly available in the Fall of 2004. The remainder of this thesis, though focusing heavily on the Beta 2 release, attempts to point out significant differences between Beta 1 and Beta 2. For illustration purposes, Figure 1 presents a screenshot of the Redwood Beta 1 environment.



Tests the running time of two sorting algorithms: quick sort and merge sort.

int		The system error code.
Timer	timer	The timer used to get the running times of the algorithms.
int	index	The loop variable for filling the array.
int*	list	The list of integers to be sorted.

```
list ← new int [ 10000 ]
```

Fills the list of integers with random values.

```
INITIALIZATION
  index ← 0
CONDITION
  index < 10000
UPDATE
  index ++
CODE
  list [ index ] ← random ( )
```

Running the quick sort.

```
timer• start ( )
quickSort ( list , 10000 )
timer• stop ( )
println ( timer• duration ( ) )
```

Running the merge sort.

Figure 1. Redwood Beta 1 User Interface Layout

1.4 Thesis Outline

The remaining chapters of this thesis discuss the Redwood programming environment in detail. Chapter 2 gives background information that helps one to understand the motivations in creating Redwood. Chapter 3 presents the initial design goals that were used to guide the development of the environment. Chapter 4 contains an overview of the main design concepts of Redwood including snippets, design trees, and drag-and-drop programming. Chapter 5 discusses in detail snippets and the Snipplet language used to generate source code. Chapters 6 and 7 provide references to the syntax for snippet displays and, respectively, to the Snipplet language. Chapter 8 includes several detailed examples of how snippets can be used and created. Chapter 9 summarizes the contributions of this thesis and presents some of our plans for future work.

Chapter 2 – Background Survey

2.1 Textual Programming Languages (TPL)

2.1.1 Characteristics of TPL

Textual programming languages (TPL) are those most often associated with programmers. Programmers typically use plain text to describe instructions to a machine. Although sometimes this source code is color coded, very little additional support is given to help visualize the meaning of code segments. Instead, programmers rely on paradigms that help encapsulate concepts. However, there are currently only two major categories of paradigms and new paradigms are rare within the industry. Programmers have been working with the two major categories of paradigms, declarative and imperative, for nearly the entire existence of computers [8].

A declarative language is one in which a programmer "describes a problem rather than defining a solution" [9]. Declarative languages are popular for solving certain types of problems, for example, problems in the area of artificial intelligence. However, imperative languages, those with which a programmer specifically defines solutions, are currently most popular. Later in this section, the strengths and weaknesses of TPL are discussed.

2.1.2 Examples of TPL

With TPL, classical programming languages come to mind. Lisp, Scheme, and ML are functional languages (on the declarative side). Prolog is a logic language, which is also declarative. Fortran, Pascal, and C fall under the "von Neumann" paradigm [8]. These are imperative languages. Examples of object-oriented languages include Smalltalk, C++, and Java. Most computer programmers are familiar with at least a few of the above-mentioned textual programming languages.

2.1.3 Strengths and Weaknesses of TPL

The most significant strength of textual programming languages is also its key weakness. Programmers know that virtually anything can be described with text. One invests time and rigor to develop products based on TPL that describe every interaction, feature, and detail. While having the potential for unequivocal precision, in human hands this method of programming leads nevertheless to mistakes, to software bugs. Programmers have long struggled with buggy code; many have realized that bugs will never be eliminated. Still, it makes one think that there might be a better way to develop software. There might be a way that takes advantage of natural human characteristics, such as the ability to quickly acquire information from images, to help reduce the burden on the programmer, to ensure some level of correctness.

The problem with text is that it is, even though learned very early on, a manmade concept. In most languages there are few natural mappings between text and physical objects [10]. As such, it is difficult to build a safety system that checks the sensibility of program statements. In a TPL it is possible to skip important details, to create non-sense, and to never realize it because with text mistakes cannot often be caught by casual inspection. It is difficult to visualize connections and interrelationships because there is no physical line being drawn between interconnected elements. Text requires more complex, human, and error-prone analysis and interpretation [11]. Besides requiring great amounts of time, the requirement for analysis and interpretation is a significant cause of buggy programs.

A large portion of time that goes into developing programs is spent during the debugging phase. Debugging is typically required for reasons related to two problems. The first problem is mistranslation from the natural language problem description to the programming language solution. The second problem is that it is often difficult to identify the connections between interdependent elements of a program and thus one has to spend time analyzing where the interdependencies exist, and where potential problems lie. Because text gives very few visual clues about correctness, and because of the human tendency to make mistakes, debugging programs created with TPL can be a tedious, imperfect, and frustrating process.

2.2 Graphical Programming Languages (GPL)

2.2.1 Characteristics of GPL

A graphical programming language is a tool that allows one to visualize and manipulate a program's components. It is common for GPL systems to use drag-and-drop manipulation, often for icon-based program representations. One can think of GPL in a broader sense however, as any graphical tool that allows one to precisely describe the components of a computer program. As such, interface builders and even program file/class trees can be thought of as small examples of GPL integrated into many development environments.

2.2.2 Examples of GPL

The development of computer graphics first and then of graphical programming systems has changed programming in particular and problem solving in general. The history of this field started in 1963 when Ivan Sutherland developed Sketchpad [12], the first interactive computer graphics application. This development opened an entirely new world of possibilities for computer programming. Computer graphics and game development took off, but twelve years had to pass before the next significant breakthrough in graphical programming occurred. Pygmalion [13], developed by David Smith, was the first icon-based programming system, the first system that started taking the shape of modern graphical programming systems. From 1975 to the present, a significant amount of work has been invested in developing graphical programming systems and visual programming environments.

At first, visual programming only worked for toy problems and many believed that it was not suitable for “real-world” projects. In her work on visual programming, Margaret Burnett discusses various approaches that have been taken to overcome what once was a rather valid perception [14]. A great amount of research went into using graphical programming for the front-end systems, specifying GUI layout. Two of the major commercial successes arising from this branch of research have been Microsoft’s Visual Basic [15] and ParcPlace System’s VisualWorks for Smalltalk [16].

Other approaches have tried to increase the range of projects that were suitable for visual programming. This was usually done by developing domain specific visual programming systems or environments. The approach varies widely and is dependant largely on the problem domain. For example, Stagecoach Software’s Cocoa (formerly KidSim) is directed towards graphical simulations and games [17], Cypress Research’s PhonePro handles telephone and voice-mail [18], Advanced Visual Solutions’ software deals with scientific visualization [19], and National Instruments’ LabVIEW focuses on laboratory data acquisition [20].

In the sequential domain, languages such as ARK, VIPR, Prograph, Forms/3, and Cube have demonstrated varied possibilities for graphical programming languages. Discussions of these and many more can be found in a survey by Boshernitsan and Downes [21] as well as in Margaret Burnett’s Visual Programming Language Bibliography [22]. In the parallel and distributed programming arena, there are several graphical programming tools that have been developed to help advance the programming capabilities of those learning the field. These tools range from analysis systems such as Pablo from the University of Illinois [23] to systems such as Paralex [24], Grade [25], and Trapper [26]. Many of these systems use similar iconic designs, while others incorporate graphs and connection-based constructs.

2.2.3 Strengths and Weaknesses of GPL

Modern GPL systems tend to be highly specialized, working well for only one type of problem. No GPL tools exist that can be used for general computer programming. Visual Basic's interface builder is good for developing interfaces. Pablo is good for writing performance analysis programs but is not designed for much else. Many other systems are technically capable for writing general programs, but are easily overwhelmed by complexity. Resulting programs have to be simple and small. With modern GPL, one must choose between specialization and ability to deal with complexity.

2.3 Hybrid Programming Languages (HPL)

2.3.1 Characteristics of HPL

Most modern GPL are actually hybrid systems in which one must interact with both graphical components and textual components. It seems like a natural extension of ideas that one should not be tied to using one method or the other. Programmers should be able to use the most appropriate method, as it is convenient. The most important aspect when dealing with HPL is to find the right balance between the amount of TPL and GPL involved.

2.3.2 Examples of HPL

Code, from the University of Texas, is one of the most apparent examples of an HPL [27]. With Code, the GPL portion is used to describe the parallelization and to give hints about the parallelization of an otherwise serial program. The system is hybrid because the serial portions of the program are written using a standard source code editor and TPL. After the serial code is written, each function is associated with an icon. These icons, in Code, can be connected to demonstrate the desired parallelized activities of the program. Compared to the tedious task of writing synchronization and message passing functionality by hand, Code seems to provide a much simpler alternative. A screenshot of the Code interface is shown in Figure 2.

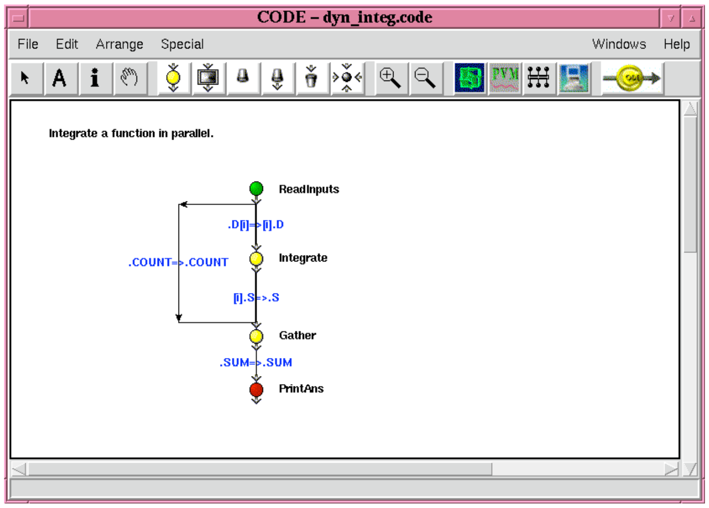


Figure 2. Code 2.0, University of Texas

2.3.3 Strengths and Weaknesses of HPL

HPL allows one to have a mixture of TPL and GPL, thus giving one the "best of both worlds." One problem with current systems, however, is that there is a fixed, and often unbalanced, mixture. The ability of a programmer not only to use both TPL and GPL, but also to select the proportion of each, is essential. Our proposed programming environment, Redwood, helps manage this balance, allowing the programmer to select the text-like components as well as the graphical components that he or she wishes to work with.

2.4 A Novel Approach for Expressing Algorithms

In his book, Tremblay says that in terms of current knowledge and technique, there are at least five levels of abstraction involved in "expressing algorithms" [28]. These are:

- Natural languages,
- Diagrams,
- Flowcharts,
- Algorithmic languages (pseudo code), and
- Programming Languages.

These levels of abstraction have changed little in the past several decades [29].

Current computer programming systems tend to require a programmer to translate from the highest level, *natural languages*, to the lowest level, *programming languages*.

Higher-level abstractions tend to be more powerful than lower-level ones, but at the cost of being less specific. In other words, in terms of describing systems, the English language is much more powerful than a programming language such as Java or C++, but programming languages have the advantage of precision. Translating from a natural language to a programming language can be quite a challenge for human programmers, as the level of precision required by programming languages is rather demanding for most humans to manage. One of the problems with the current approach is that there are certain aspects of any program that need to be very precise, and certain aspects that can be specified at higher-levels of abstraction. Paradigms such as object-oriented programming address these needs to some extent.

With an object-oriented programming language, one can specify, with a relatively high level of abstraction, that one requires access to an object, which belongs to a certain category (or *class*) of objects. One can also specify, on a lower level of abstraction, the precise calculations needed to manipulate an object of the class. Besides being highly dependent on the skills of the programmer, there are several weaknesses with this approach. The primary weakness is that, through the use of textual programming languages, one cannot represent an object as it appears in the physical world. The syntax of any textual programming language, even though familiar and seemingly fluent for many programmers, is arbitrary, and often unrelated even to the arbitrary structure of natural languages.

An ideal method of computer programming would use the precise level of abstraction that is needed to most efficiently express the desired instructions. That is, each of the five levels of abstraction, as mentioned above, would be used appropriately.

Much in the same way that interface builders help design interfaces, future programming system environments will allow programmers to graphically manipulate and interconnect programming instructions using natural language, diagrams, flowcharts, pseudo code, programming languages, and related multimedia representations where suitable.

Chapter 3 – Overview of the Redwood Proposal

3.1 Design Goals

Redwood came about as part of an amalgamation of ideas put together into one project. Initially, in Spring 2003, a design goals document was completed, listing several key directions for developing a new IDE: enhanced support for hierarchical design, drag-and-drop manipulation, algorithmic independence, object-oriented programming, parallelism, usability, open source code, and software documentation [30]. Each of these foci, as discussed below, will be pivotal in the future of programming.

3.1.1 Hierarchical Design

Designing from the top-down or bottom-up should be more recognizable throughout the code writing/maintaining process. Dependencies and relationships could be demonstrated through graphical representations. This should also be carried out with an integrated versioning system that will keep track of build numbers, developer/change information, and allow for multiple concurrent versions of portions of a project. This should also include collapsible/expandable code, allowing developers to distinguish the big picture ideas of a program from its smaller details and ideas.

3.1.2 Drag-and-Drop Manipulation

Because most new code is structurally similar to code that has been used in the past, templates and some elements of drag-and-drop programming should be employed to increase programmability and decrease debugging time. Templates should be available from built-in libraries, user libraries, and in online collections (available directly in the development interface).

3.1.3 Algorithmic Independence

Where possible, entire algorithms should be interchangeable, simply by dragging and dropping, without modifying code. By replacing a single element in the graph structure of a program, one can incorporate more efficient implementations of an algorithm. This allows for fast conversions from rapid prototypes to full-scale products. Once again, algorithms, such as those for sorting, should be available in built-in libraries, user libraries, and in online collections (which can be updated with submissions from other developers). A priority ranking system should be made available for such algorithms, giving developers the ability to see (in graphical form), the ratings for the following categories: speed, memory use, parallelizability, and price (for proprietary algorithms). Such rankings could be maintained by groups of developers (through a voting system) giving developers the ability to prioritize, select, and immediately plug in, new algorithms.

3.1.4 Object-Oriented Programming

Objects, methods, and data could be organized using principles from Java. Packages group related objects into a hierarchy of components. Unlike in Java however, access control should be more strictly guarded, with permissions (at the object/package design level) for reading, writing, extending/overriding, implementing, and so forth. Enforced by both the language and in some cases a layer of encryption (to be used with proprietary algorithms), developers should be ensured that their code is used correctly within the development environment.

3.1.5 Parallelism

Designing for single processor, multi-processor, and distributed processor machines (with various configurations) could be done virtually automatically when appropriate hints are given to a compiler. These hints could be given and represented in graphical form, giving an intuitive flow to programs. This could include allowing a user to deploy a program optimized to run on a specific cluster or machine (by specifying machine topology and networking layers via connection graphs), or to be ready for general use.

3.1.6 Usability

In all of the above, the design environment should be immediately familiar for virtually any programmer to use – without the need for extensive training. Based on ideas from

C/C++, Java, Perl, and a few other languages, the code level (which may not be reached by all developers), should be relatively simple to learn. Much of the overhead (such as classes and templates) should be taken care of by the graphical environment, so knowledge of syntax and other implementation details could be unnecessary.

3.1.7 Open Source Code

These software design ideas should be applicable to the open and shared source initiatives, but should also be advantageous to internal (within a company) settings, as code sharing would become more fluid.

3.1.8 Software Documentation

The crux of shared code is the ability to understand it without needing to take an in-depth look at the code. Shared source code should make clear through comments and naming conventions, what a particular function does for instance. Design principles, such as commenting could be enforceable in the development environment (at the option of the developer or employer) and have various levels of enforcement. Comments, for example, could be required before a function is written, after a function is completed, before a build can proceed, or before the CVS can be updated. These settings should be applied, and possibly locked, by employers so that employees must comply with the standards, or left open for a developer that needs more freedom. Such settings could be local to the computer or available on the network.

3.2 Current Status

The above goals have driven the development of Redwood. Several of the goals were accomplished in Beta 1, completed in September 2003. Additional goals, which provide significant enhancements to the overall project, are largely completed for the Beta 2 version, which will be publicly released in the Fall of 2004. Still, there are additional, more specialized goals to complete during the future development of Redwood. A synopsis of the environments goals, listed in alphabetical order, and their achievement statuses is shown in Table I.

Table I Major Redwood Project Goals and Their Statuses

Goal	Beta 1 (2003	Beta 2 (2004)	Future (2005/06)
Advanced Support for Parallelism			✓
Collapsible/Expandable Code	✓	✓	✓
Commenting/Design Enforcement			✓
Data Security			✓
Data Security – Encryption Layer			✓
Dependency Highlighting			✓
Drag-and-Drop Programming	✓	✓	✓
Dynamic Templating System		✓	✓
Easy-to-Learn Environment	✓	✓	✓
Familiar Meta-Language		✓	✓
General Support for Parallelism			✓
Integrated Online Template Library			✓
Integrated Versioning System			✓
Integrated Voting/Ranking System for Snippets			✓
Package Organization		✓	✓
Ready for Open/Shared Source Development		✓	✓
Support for Whole Algorithm Replacement			✓
Templating System	✓	✓	✓

Chapter 4 – Main Design Concepts

4.1 Snippets

Snippets are the cornerstone of everything that is built within the Redwood environment. They provide, in combination with design trees, the structure of a program. Snippets are available for each major aspect of a programming language: functions, loops, statements, and so forth. Unlike with a TPL, the grammar for a language is not predefined with Redwood. Instead, the language is constructed from the set of tools that one has available. These tools are 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 small part” of a solution. A snippet is comprised of two sections, the display section and the template section. The structure of a snippet is described using XML. The *display section* of a snippet describes the visualization of the snippet. It specifies how the elements of the snippet are placed on the screen. The elements of a snippet are called *snippet editors*. Several snippet editors, including a `CodeEditor` and `IdentifierEditor` are included in Redwood. These editors allow programmers to store values and structures, including other snippets, inside of snippets. They can represent anything from a block of code to a single numeric value. The XML syntax for the display section of a snippet is discussed in detail in Chapter 6.

The template section of a snippet is made up of one or more XML *tags* that contain *Snipplet scripts*. A *Snipplet script* is a script written in the Snipplet language for the purpose of mapping from the visual structure of a snippet into the programming language that the user has selected for output. In other words, a Snipplet script (often referred to as a template) generates source code. The Snipplet language is presented in detail in Chapter 7.

4.2 Design Trees

Design trees are constructed when snippets are nested. A design tree is the structure that allows, for example, a function to contain a block of code and a block of code to contain statements. This concept may not seem readily novel or useful to distinguish. In a standard compiler, a design tree would be called a parse tree. The difference between these two concepts however, is important. A parse tree is composed of text built from static rules based on a grammar. A design tree, powered by snippets, however, is built with dynamic rules. It is as if one were to have a programming language that was able to adapt to the needs of the programmer. The concept of a design tree provides the framework through which snippets are connected.

During the build process, a Snipplet script can access a program's design tree using two methods. First, the script can use the `$snippetGraph` and `$rootSnippetGraph` variables which provide direct access to snippets and snippet editors. Second, the script can use snippet editor function calls and filters. Both methods are discussed at length in Chapter 7.

4.3 Drag-and-Drop Programming

Drag-and-drop programming is an aspect of virtually all GPLs and HPLs. In one of the most representative examples, Carnegie Mellon's educational tool Alice [1], drag-and-drop programming was demonstrated more as textual programming combined with drag-and-drop to help reduce the burden on the programmer. The programmer would no longer have to remember the exact syntax for a `for loop` or an `assignment` statement. He/she only has to remember how pieces could fit together. Other GPL, drag-and-drop programming systems, similar to Alice, fall into two categories: the highly specialized, or the trivial. Alice was itself specialized, providing an interface to programming for students. However, taking the way that drag-and-drop programming was used in Alice one step further leads to a method that offers graphical, drag-and-drop capabilities that can be used for general-purpose computer programming. This was a large part of the inspiration for Redwood.

The drag-and-drop elements of Redwood are similar to those in Alice. That is, constructs are selected from a menu and dragged into place in an editor. Once in place, constructs can be edited and repositioned as necessary. One can create a program that contains a class. Instead of typing in the syntax elements as one would in a traditional TPL, Redwood allows one to drag an entire `class` construct. Then, the programmer fills in the name and other customized aspects of the class. Similarly, one can drag in a function, a `for loop`, or an `assignment` statement. As learned from Alice and other GPL/HPL systems, drag-and-drop programming alone is not enough to create a major change in programming. However, drag-and-drop programming, when combined with the power of snippets, with the ability to extend one's language and present code in more meaningful ways, may.

4.4 Visualization Details

The visualization details immediately noticeable in Redwood Beta 2 may seem relatively insignificant compared to those in Beta 1. Through the development of Beta 1, it was realized that space usage could become a problem. Textual programs take up a fair amount of screen real estate. With Beta 1, programmers found themselves being weighed down by the amount of space simple constructs took up. For example, in Beta 1, because of the linear layout requirements, a basic `for loop` was unnecessarily large, even if it contained only a single statement.

In Beta 2, one of the motivating factors for rewriting the snippet handling code was space (and organization in general). In TPLs, programmers were used to working with loop constructs that took minimal space. Even with the ability to hide portions of code in Beta 1, it still was not nearly enough; at issue were the snippets themselves. The snippet visualization in Beta 1 was performed in a way that resulted in large graphics. Large outlines surrounded each snippet and the snippet editors. With the Beta 2 release, the visual structure that allows one to modify a snippet's properties, such as visibility, is hidden until necessary. The programmer uses the mouse to bring up the controls. When the mouse cursor runs over a snippet, the controls become visible. For convenience, holding the `escape` key will maintain the state of the controls so they do not disappear when moving over other objects. Similarly, releasing the `escape` key will force the controls to disappear.

The controls used in Beta 1 for adjusting the visibility of components within the snippet were *disclosure triangles* and *disclosure dots*. As of the current status of Beta 2, only *disclosure triangles* (as found in Mac OS X) are present. The future of *disclosure dots* is currently undecided. For reference, a *disclosure triangle* is a triangular shape that when pointing right is non-disclosing, meaning the contents associated with that disclosure triangle are invisible. When the triangle points downward the contents are disclosed. *Disclosure dots* provide finer-grained controls over visibility allowing certain subcomponents to be made visible or invisible. In Beta 2 an alternative method for this with a more natural mapping or at least a more standardized approach will be introduced.

Chapter 5 – Snippets and the Snipplet Language

5.1 More on the Snippet Concept

Snippets are tools for creating mappings between visual representations of programming constructs and source code output. The visual part of a snippet, its display section, is a collection of snippet editors that are organized in two dimensions using a table layout. These editors can be used to store other snippets, display data, and/or act as inputs. Snippet editors are standard Java Swing components that inherit from the `SnippetEditor` class. Each snippet editor has a name and a type. The built-in snippet editors include `CodeEditor`, `DropdownEditor`, `ExpressionEditor`, `HR`, `IdentifierEditor`, `LineEditor`, `ParametersEditor`, `StringEditor`, and `TypeIdentifierEditor`. This list will be expanded in the future because the current set is minimal and more powerful tools can be created that allow one to more efficiently visualize and manipulate programming constructs. The name of a snippet editor is assigned in the display section of its containing snippet. Using the name of a snippet editor inside a Snipplet script, one can access the editor and its contents. This is discussed in detail in Chapter 7.

The template section of a snippet stores *Snipplet scripts*. For each language supported by a given snippet, at least one template has to be created. More than one template can be created for a single language and more than one language can be used by a single template. This is described in Chapter 8. For each language, the templates for that language are concatenated in-order. Thus, the result is one Snipplet script per language for each snippet. The portion of a Snipplet script that does not appear to exist within a function is actually within a function called "template". This function can be accessed as any other function. Functions placed inside the template functions are accessible in two ways. The first method is by directly using a function call. Many Snipplet scripts can use "helper" functions to achieve their mappings. The second method is to use a snippet editor function call. The exact syntax of a snippet editor function call is shown in Figure 9, Production 45. A Snipplet can access the functions of another Snipplet via this syntax. This is one method of accessing the design tree. For additional details, the reader is again referred to Chapters 7 and 8.

5.2 Snippet Structure

Before rushing into creating one's own snippets, one should survey the landscape of existing snippets. This is simple to do because snippets are organized by category in their hierarchical structure. In addition, when the online library is integrated it will be fully searchable and viewable by category as well. A developer must decide what type of snippet is to be created. For example, if one is implementing a new type of operator, then the type is most likely an `Expression` and/or `Statement`. The `extends` attribute allows one to formally specify which type of snippets a new snippet is similar to. This is discussed in more detail in Chapter 8.

5.3 Snippet Development Stages

A snippet is defined and used in a process that can be represented, with certain simplifications, as shown in Figure 3. (Note that activities are represented using shaded rounded rectangles, while the results of these activities are shown using regular rectangles.)

First, the definition of the snippet is created using XML code. Next, at the environment's start-up the `snippet` tag is listed in Redwood's `Snippet Chooser` tool, thus the snippet becomes available for use ("selectable," or ready to be dragged-and-dropped). Then, if needed in the program, when the snippet is dragged into the program's workspace, the graphical representation of the snippet is rendered on the screen (*visualized snippet*). Because in practice two types of snippets are used *(base snippets*, or *generic snippets*, such as a generic `class` snippet and *customized snippets*, or *instantiated snippets*, such as a `book class` snippet), the next typical step in using snippets is customization, where the details of the snippet are fleshed out (e.g., a generic `class` snippet becomes the instantiated `book class` snippet). Finally, when the current project is built, the snippet is mapped onto its associated code according to the templates section of the snippet definition. The snippet development stages are illustrated in Figure 3. Note that for simplicity some details of the snippet development stages are omitted.

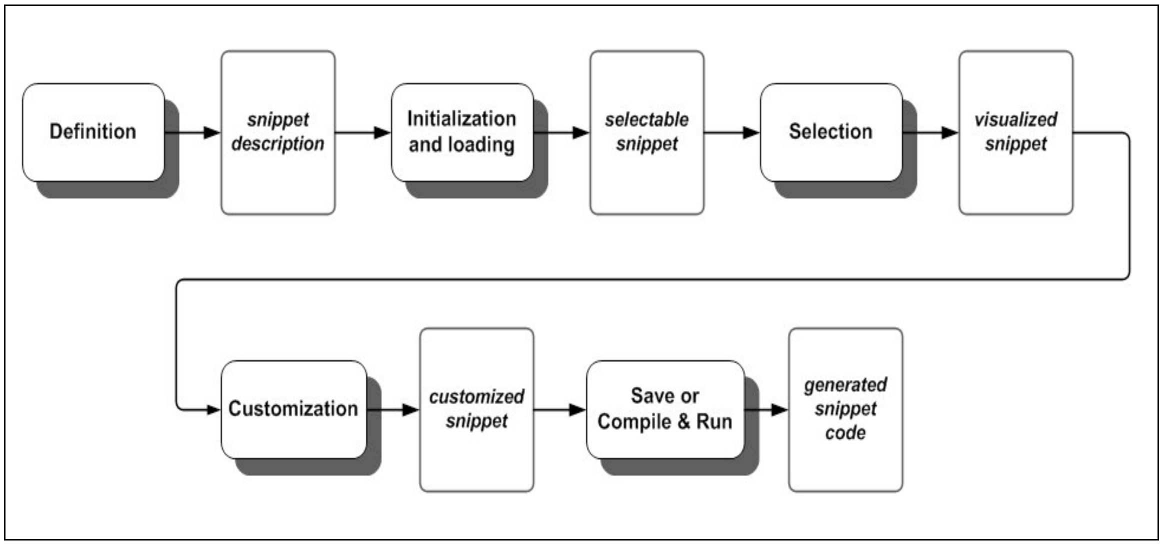


Figure 3. Snippet Development Stages

5.4 Snippet Display Syntax and the Snipplet Language

A snippet's display and templates are described using the snippet display syntax (SDS) and Snipplet language (SL), respectively. The display section of a snippet describes the portions of the snippet that are visible to the user and the template section describes the mapping from the visualized snippet to the programming language output (generated code). Displays are currently described using static XML and templates are described using dynamic Snipplet scripts. In a future release, displays may also be dynamically created and modified.

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 for Redwood, the interface builder. Upon examining the source files of Redwood Beta 2, one may notice that several of the files end with the "interface" file extension. These files are used to describe interfaces, such as for dialog boxes, and are constructed using XML. Interface files are a convenient way to build interfaces without 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. This technology becomes especially important when considering localized versions of Redwood. Perhaps more importantly, the technology is flexible enough to be reused for external projects. The interface builder allows one to load an interface dynamically from a file so the named components are saved as variables accessible via a returned hash map. Chapter 6 provides a detailed reference for the syntax related to creating snippets.

The Snipplet language is designed 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 languages in mind, PERL [31] and JavaScript [32]. 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, Snipplet makes use of variant types and has several built-in functions specifically designed for parsing. Unlike with PERL and JavaScript however, the language set is kept to a minimum. Even I/O is not included. Another important factor that went into designing the Snipplet language was security. Eventually an online snippet library will be integrated into the Redwood environment. When programmers download new tools, it is essential that they do not have to worry about viruses. If a language has too many features, it could become easily exploited for malicious purposes. Chapter 7 provides a detailed description of the Snipplet language.

Chapter 6 – Snippet Display Syntax (SDS)

6.1 SDS Features

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

In Beta 1, only vertical, linear layouts were supported. In Beta 2, the snippet display tag contains a 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 in Beta 2 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. Unlike HTML tables however, the Beta 2 table layouts only allow one element (which can be a nested table) per cell.

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 (these will be discussed in more detail in Section 8.2). 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.

6.2 SDS Reference

The following reference assumes that one has a basic knowledge of XML including knowledge of tags and attributes. All possible tags, along with several examples of snippet editors, are listed below. It is important to note that snippet tags are case-sensitive as many of them refer to Java class names.

6.2.1 SDS Tag – display

The `display` tag is used as a wrapper for the entire snippet display. It contains either a snippet editor or, more commonly, a table.

6.2.2 SDS Tag - table

The `table` tag organizes cells used for storing snippet editors and nested tables into rows and columns. A table may only directly contain `tr` (table row) tags. Table II gives the list of attributes available to the `table` tag, their descriptions, and also gives an example using the `table` tag.

Table II The SDS table Tag: Attributes and Example

Attribute	Description
`margin`	The size of the margin placed around the outside of the table. This gives padding to the contents.
`margin-left`	The size of the left margin. This is useful for creating a "tabbed" look.
`margin-right`	The size of the right margin.
`margin-top`	The size of the top margin.
`margin-bottom`	The size of the bottom margin.

Example

```
<table margin="2">
   <tr>
      <td>
         <LineEditor>
            <param name="Text">
               <String value="Line Editor 1"/>
            </param>
         </LineEditor>
      </td>
      <td>
         <LineEditor>
            <param name="Text">
               <String value="Line Editor 2"/>
            </param>
         </LineEditor>
      </td>
   </tr>
</table>
```

6.2.3 SDS Tag - tr

The `tr` tag represents a table row. A `tr` tag has no attributes and may only directly contain `td` (table data) tags. See the example given in Table II for use of the `tr` tag.

6.2.4 SDS Tag - td

The `td` tag represents a cell (td stands for table data). A `td` tag may directly contain table or snippet editor tags. Unlike HTML, there is no need for a `td` tag to contain anything. The appropriate amount of space is allocated for the cell whether or not the cell contains anything. Table III lists the attributes and gives an example using the `td` tag.

Table III The SDS td Tag: Attributes and Example

Attribute	Description
`width`	The width of the column, which can be an absolute (pixel) value or a percentage of the table width. The percentage is the percentage after statically allocated widths are used. The default width is the preferred width of the largest visible component. The actual width of a column will be the largest of the specified widths if multiple widths are specified for a single column.
`height`	The height of the row, allocated similarly to the width.
`colspan`	The number of columns over which the cell exists (spans). The default is 1.
`rowspan`	The number of rows over which the cell exists. The default is 1.
`halign`	The horizontal alignment of the contents of the cell. The default is "fill". Possible values include "left", "right", and "center".
`valign`	The vertical alignment of the contents of the cell. The default is "fill". Possible values include "top", "bottom", and "center".

Example

```
<tr>
   <td width=”100%” colspan=”3”>
      <LineEditor>
         <param name="Text">
            <String value="Line Editor 1"/>
         </param>
      </LineEditor>
   </td>
</tr>
<tr>
   <td width=”50%”>
      <LineEditor>
         <param name="Text">
            <String value="Line Editor 2"/>
         </param>
      </LineEditor>
   </td>
   <td width=”12”></td>
   <td width=”50%”>
      <LineEditor>
         <param name="Text">
            <String value="Line Editor 3"/>
         </param>
      </LineEditor>
   </td>
</tr>
```

6.2.5 SDS Snippet Editors

Various snippet editors are included in Redwood. This reference describes how to use snippet editors in general and gives several specific examples. To use a snippet editor, give the tag the name of the snippet editor class file. This is case-sensitive. For example, to use the `LineEditor` class, one would use `<LineEditor/>`. The `param` tag is used for setting parameters in snippet editors. The `function` tag is used to call functions on the snippet editor. The `function` tag can be used in place of the `param` tag, but for readability, this is generally not recommended.

6.2.6 SDS Tag - param

The `param` tag is an alias for the `function` tag with a name parameter that must start with "set". For example, if a `param` tag has the name "Enabled", the equivalent `function` tag would have the name "setEnabled". See the `function` tag reference for additional details.

6.2.7 SDS Tag - function

The `function` tag allows one to call any public function of the snippet editor. The tag requires the `name` attribute and may contain any number of *value tags*, as discussed next. Table IV contains the list of attributes that may be used with the `function` tag. Only the `name` attribute may be used with the `param` tag.

Table IV The SDS function and param Tags: Attributes and Example

Attribute	Description
`name`	The name of the function to be called on the snippet editor.
`type`	The return type of the function.
`cast`	The type to cast the result value to.

Example

```
<td><LineEditor>
   <param name="Text"><String value="Line Editor 1"/></param>
   <param name="MinimumSize">
```

Table IV Continued from previous page

`      <Dimension>`
`         <param name="Size">`
`            <int value="-1"/>`
`            <int value="10"/>`
`         </param>`
`      </Dimension>`
`   </param>`
`</LineEditor></td>`
`<td><LineEditor>`
`   <function name="setText">`
`      <String value="Line Editor 2"/>`
`   </function>`
`</LineEditor></td>`

6.2.8 SDS Value Tags

Various value tags can be used within `param` and `function` tags. These currently include the following types: nested `function` tags, `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`, `String`, and any other type in any of the following packages: the default package, `redwood`, `redwood.plugins`, `java.lang`, `java.awt`, `java.awt.event`, `java.util`, `javax.swing`, or `javax.swing.event`. The `cast` attribute (as in the `function` tag) can be used for most of the value tag types. For classes without default constructors, the `constructor` tag may be used to specify the initialization method of the object (see the "Foreground" `param` tag below). Table V gives an example using several types of value tags.

Table V The SDS Value Tags: Attributes and Example

Attribute	Description
various	Most value tags can use the `cast` attribute, some can use the `value` attribute.
Example	

`<LineEditor>`
`   <param name="Text"><String value="Line Editor 1"/></param>`
`   <param name="Editable"><boolean value="false"/></param>`
`   <param name="Foreground">`	

Table V Continued from previous page

```
      <Color>
         <constructor>
            <int value="128"/><int value="128"/><int value="128"/>
         </constructor>
      </Color>
   </param>
   <param name="MinimumSize">
      <Dimension>
         <param name="Size">
            <int value="-1"/>
            <int value="10"/>
         </param>
      </Dimension>
   </param>
</LineEditor>
```

In the following, examples are provided for a few of the included snippet editor types. This list is not all-inclusive. For additional information about specific snippet editors, one should read the generated HTML documentation that will be included with the Redwood download when publicly available at [33].

6.2.9 SDS Tags - LineEditor, StringEditor, IdentifierEditor, TypeIdentifierEditor

A `LineEditor` displays and/or allows the user to input a single line of text. A `StringEditor` is similar to a `LineEditor` except that it allows for multiple lines of text. `IdenfitierEditor` and `TypeIdentifierEditor` are special types of `LineEditors` that are designed for users to enter identifiers (i.e. variable names) and type identifiers (i.e. class names). In the future, these modified types will provide enhanced usability features such as auto-completion for known variable names. Table VI gives a usage example and the list of parameters commonly used with these editors.

Table VI The SDS LineEditor and StringEditor Tags: Parameters and Example

Parameter	Description
`Editable`	Whether or not the text is editable. Use a `boolean` value.
`FontSize`	The size of the font to be displayed. Use an `int` value.
`Foreground`	The color of the text. Use a `Color` value.
`Text`	The text to be displayed. Use a `String` value.

Example

```
<LineEditor name="id">
   <param name="Text"><String value="Line Editor 1"/></param>
   <param name="Editable"><Boolean value="true"/></param>
   <param name="Foreground"><Color>
      <constructor>
         <int value="255"/>
         <int value="0"/>
         <int value="0"/>
      </constructor>
   </Color>
</LineEditor>
```

6.2.10 SDS Tag - CodeEditor

A `CodeEditor` allows one to place many snippets in a vertically linear order. These are used in the `SnippetGroup`, `Class`, and `Function` snippets, among others, for dropping code. Table VII gives the parameters that are commonly used with the `CodeEditor` tag and gives an example.

Table VII The SDS CodeEditor Tag: Parameters and Example

Parameter	Description
`Allow`	Used to specify which types of snippets can be dropped. Use a `LinkedList` of `String` values.
`ConstSnippets`	Whether or not new snippets can be added to / current snippets can be removed from the editor. Takes no parameters, used to set only.

Example

```
<CodeEditor name="code">
   <param name="Allow"><LinkedList>
      <String value="Comment"/>
      <String value="Statement"/>
   </LinkedList></param>
</CodeEditor>
```

Chapter 7 – The Snipplet Language (SL)

7.1 SL Features

The Snipplet language 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 [31] is often used for CGI programming [34]. 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 [32]. 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.

The most important aspects of the language design are the functionality of `return` statements and the scalability of variables. The `return` statement, unlike in most other languages, does not automatically halt function execution. Instead, return values are concatenated. This is effective for source code generation in that the repetitive construction of functions can be reduced. To jump out of a function after a return or at any other time, a `break` statement can be used. Like 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 important 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 languages. At the same time, the language should be powerful enough and flexible enough to simplify the often complex demands involved in source code generation. Although uncommon with respect to other languages, the issue of concatenating return values became important as, without it, for virtually every function that one would write, one would need to create an output variable specifically used for concatenation.

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 one needs to know to work with it are relatively short.

7.2 SL Grammar

In order to facilitate the implementation of the Snipplet interpreter, a custom *grammar* and *parser generator* were created. A grammar file describes language syntax. In the case of Snipplet scripts, everything revolves around templates. A *template* is a single Snipplet script that can be interpreted and executed. Once a grammar is constructed, a parser can be created for the grammar. The parser generator used for Redwood takes the Snipplet grammar and produces such a parser.

The Snipplet parser is a Java class whose `parse` function takes a string as a parameter. This string is parsed and the result is placed into a *parse tree*. The parse tree represents a full analysis of the syntactical structure of the string. It breaks the string down so that one can determine that the string is a template, the template contains blocks, the blocks contain statements, and so on. Given a parse tree, another Java class, `SnippletHandler`, is used to interpret the script. At this point, the result of the script, specifically the concatenated return values of the script, are returned.

The Snipplet grammar, presented below in a variant of Extended Backus-Naur Form (EBNF), is separated by level for easier reading. Levels 0, 2, and 3 are the most significant, containing the start symbol and the types (and specifications) of statements.

Level 0, also known as the “start symbol”, is shown in Figure 4 and is the wrapper for the entire template code; it encapsulates all Snipplet code.

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

Figure 4. SL Grammar: Level 0 (the Start Symbol) of the SL Grammar

In Figure 4, one should notice that allowance of whitespace is being explicitly specified. Figure 5 contains the basic pieces of a template, blocks (or collections of statements) and whitespace (which includes C++ style comments). The syntax within single quotes follows PERL regular expression syntax.

```
2       block ::= statement*

3       WS ::= (‘\s+’ | ‘//.*’ | ‘/\*([\u0000-\u0029\u002B-\u9999]|\*[\u0000-\u002E\u0030-
        \u9999])*\*/’)*
```

Figure 5. SL Grammar: Level 1, Basic Template Components

Figure 6 shows Level 2 of the SL Grammar. This level is one of the most significant as it describes the different types of statements that are possible with the Snipplet language. A statement is the crux of the grammar; each element of the script can be reduced to a statement.

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

Figure 6. SL Grammar: Level 2, Statement Types

Level 3 is shown in Figure 7. This level gives a detailed description for the specification of each of the statement types listed in Level 2 (Figure 6). For convenience, the `expression` production is broken down into sub-expressions (starting with `binaryArithmeticLevel5`). Though these sub-expressions should belong to higher levels, they have been grouped together for more efficient organization.

```
break ::= 'break'

declaration ::= 'var' SP identifier (WS '=' WS (arrayInitializer | expression))?

expression ::= binaryArithmeticLevel5

//Sub-expressions

binaryArithmeticLevel5 ::= binaryArithmeticLevel4 binaryArithmeticLevel5a?

binaryArithmeticLevel5a ::= WS '\|\|' WS binaryArithmeticLevel4
binaryArithmeticLevel5a?

binaryArithmeticLevel4 ::= binaryArithmeticLevel3 binaryArithmeticLevel4a?

binaryArithmeticLevel4a ::= WS '&&' WS binaryArithmeticLevel3
binaryArithmeticLevel4a?

binaryArithmeticLevel3 ::= binaryArithmeticLevel2 binaryArithmeticLevel3a?

binaryArithmeticLevel3a ::= WS '!=|<|<=|==|>=|>' WS binaryArithmeticLevel2
binaryArithmeticLevel3a?

binaryArithmeticLevel2 ::= binaryArithmeticLevel1 binaryArithmeticLevel2a?

binaryArithmeticLevel2a ::= WS '\+|\-' WS binaryArithmeticLevel1
binaryArithmeticLevel2a?

binaryArithmeticLevel1 ::= binaryArithmeticLevel0 binaryArithmeticLevel1a?

binaryArithmeticLevel1a ::= WS '\*|/|%' WS binaryArithmeticLevel0
binaryArithmeticLevel1a?

binaryArithmeticLevel0 ::= mainExpressionPart

//End sub-expressions

return ::= 'return' SP expression

functionDeclaration ::= 'sub' SP identifier WS '\{' WS block WS '\}'

if ::= ifPart (WS elseIfPart)* (WS elsePart)?

loop ::= dowhileLoop | foreachLoop | forLoop | whileLoop
```

Figure 7. SL Grammar: Level 3, Statement Specifications

Level 4, shown in Figure 8, is a support level. A few important language features are specified here. Perhaps most significant is the forced use of curly braces for `if statements` and `loops`. Such forced formatting helps to maintain the ease of distinction between Snipplet code and output code.

```
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 '\}'
```

Figure 8. SL Grammar: Level 4, Primary Support for Statements

Figure 9. shows the productions for Level 5 of the SL grammar. The most important language feature apparent in this level is the use of multi-line strings (similar to those used in PERL) which are, like forced formatting, a useful tool in helping to distinguish between Snipplet and output code. Although automatic formatting will be implemented in a future release of Redwood, Snipplet programmers may wish to see roughly what the output will look like before running their programs.

```
37      arrayIdentifier ::= scalarIdentifier ('\[' expression? '\]')+

38      assignment ::= identifier WS ('=' | '\+=' | '\-=' | '\*=' | '/=' | '%=') WS
        expression

39      doubleValue ::= '(\+|\-)?[0-9]+\.[0-9]+'
```

Figure 9. SL Grammar: Level 5, Secondary Support for Statements

```
functionCall ::= identifier WS '\(' WS parameters? WS '\)'

hashIdentifier ::= scalarIdentifier '\{' expression? '\}'

longValue ::= '(\+|\-)?[0-9]+'

parentheses ::= '\(' expression '\)'

scalarIdentifier ::= '[\$@A-Za-z_][A-Za-z_0-9]*'

snippetEditorFunctionCall ::= '\[#' WS expression (WS ':' WS parameters)? WS '\]'

string ::= '"(\\"|[\u0000-\u0021\u0023-\u9999])*"|\'(\\\'|[\u0000-\u0026\u0028-
\u9999])*\''

unaryArithmetic ::= '!' WS expression

condition ::= '\(' WS expression WS '\)'
```

Figure 9. Continued from previous page

Level 6, shown in Figure 10, is a minor level that contains the specification for function parameters. Similarly to PERL, as many parameters as are needed may be sent to a function. Unlike PERL however, the Snipplet language does not collapse array or hash tables into the parameters list.

```
parameters ::= expression | expression (WS ',' WS expression)+
```

Figure 10. SL Grammar: Level 6, Support for Functions

7.3 SL Function Reference

The following is a list of the built-in functions that can be used in the Snipplet language. This reference is provided to give greater depth in understanding how Redwood is able to map from programming construct visualizations to target languages.

7.3.1 SL Function - arrayLength

The `arrayLength` function gets the length of the specified array. Table VIII describes the function's parameters and return value, and gives an example of how the function can be used.

Table VIII The SL arrayLength Function: Description and Example

Parameter	**Description**
Array	The array whose length is to be returned.
Return	**Description**
Integer	The number of elements in the array.

Example
`var exampleArray = (1, 2, 3, 4, 5);`
`return arrayLength (exampleArray);`	

7.3.2 SL Function - datatype

The `datatype` function determines the data type of the specified object. The parameter to this function, as shown in Table IX, can be of any type. Table IX provides an example and lists the parameters and return types for the `datatype` function.

Table IX The SL datatype Function: Description and Example

Parameter	**Description**
Any Type	The object whose datatype is to be returned.
Return	**Description**
String	The datatype of the specified object. Possible return values include: "Array", "Associative Array", "Integer", "Real", "String", and "Snippet".

Example
`var exampleArray = (1, 2, 3, 4, 5);`
`return "Datatype #1: " + datatype (exampleArray) + "\n";`
`return "Datatype #2: " + datatype (exampleArray[0]);`	

7.3.3 SL Function - defined

The `defined` function determines whether or not the specified object is defined. Table X provides a description and example for the `defined` function.

Table X The SL defined Function: Description and Example

Parameter	**Description**
Any Type	Determines whether or not the specified object is defined.
Return	**Description**
Integer	1 if the object is defined, 0 otherwise.

Example
`if (defined (checkString)) { return “checkString is defined.\n”; }`
`else { return “checkString is not defined.\n”; }`	

7.3.4 SL Function - evaluate

The `evaluate` function dynamically evaluates the specified Snipplet code. The string parameter, as shown in Table XI, can be a dynamically generated string containing Snipplet code.

Table XI The SL evaluate Function: Description and Example

Parameter	**Description**
String	Snipplet code to be dynamically evaluated.
Return	**Description**
Any Type	The result of the evaluation of the specified code.

Example
`var codeString = “var x = 5; return x;”;`
`return evaluate (codeString);`	

7.3.5 SL Function - getSnippetType

The `getSnippetType` function returns the type of the specified snippet. Table XII describes and provides an example for `getSnippetType`.

Table XII The SL getSnippetType Function: Description and Example

Parameter	**Description**
Snippet	The snippet whose name is to be returned.
Return	**Description**
String	The name of the specified snippet.
Example `var codeString = getSnippetType ($snippetGraph{“”});`	

7.3.6 SL Function - getUID

The `getUID` function returns a unique identification number. This number is unique only to the build and is not guaranteed to be stable. This function is generally used to for generating unique temporary variable names. Table XIII describes the return value and gives an example for the `getUID` function.

Table XIII The SL getUID Function: Description and Example

Parameter	**Description**
none	n/a
Return	**Description**
Integer	A unique identification number.
Example `return “int temp_” + getUID () + “ = 0;\n”;`	

7.3.7 SL Function - instanceOf

The `instanceOf` function checks to see if a snippet is or extends the specified type. Table XIV provides a description and example.

Table XIV The SL instanceOf Function: Description and Example

Parameter	**Description**
Snippet	The snippet to be checked.
String	The type of snippet to check for (including with extends attribute).
Return	**Description**
Integer	1 if the snippet is an instance of the specified type, 0 other wise.

Example
`var codeSnippets = $snippetGraph{“code”};`
`var graph = codeSnippets[1];`
`if (instanceof (graph{“”}, “Expression”))`
`{`
`   return [#“code”];`
`}`	

7.3.8 SL Function join

The `join` function combines the specified array with the specified delimiter. Table XV describes the `join` function and provides an example.

Table XV The SL join Function: Description and Example

Parameter	**Description**
String	The delimiter with which the elements of the array are to be joined (the delimiter will be placed between each adjacent pair of elements in the array).
Array	The array whose elements are to be joined.
Return	**Description**
String	The joined array whose format is like: element1delimelement2…

Example
`var exampleArray = (1, 2, 3, 4, 5);`
`return join (“, ”, exampleArray);`	

7.3.9 SL Function - keys

The `keys` function returns the keys of the specified associative array (hash table). Table XVI contains an example and description of the function parameters and return value.

Table XVI The SL keys Function: Description and Example

Parameter	Description
Associative Array	The associative array whose list of keys is to be returned.
Return	**Description**
Array	The list of keys associated with the specified associative array.
Example	

```
var associativeArray = (0, “Zero”, 1, “One”, 2, “Two”);
var sum = 0;
foreach var key (keys (associativeArray))
{
   return key + “\n”;
   sum += key;
   return sum + “\n”;
}
```

7.3.10 SL Function - removeElementFromArray

The `removeElementFromArray` function removes the specified element from the specified array. Table XVII describes and provides an example for this function.

Table XVII The SL removeElementFromArray Function: Description and Example

Parameter	Description
Array	The array from which an element is to be eliminated.
Integer	The index of the element to be eliminated.
Return	**Description**
none	n/a
Example	

```
var exampleArray = (1, 2, 3, 4, 5);
removeElementFromArray (exampleArray, 2);
return join (“, ”, exampleArray);
```

7.3.11 SL Function - replaceAll

The replaceAll function, described in Table XVIII, replaces the specified regular expression delimiter in the specified string with a specified value.

Table XVIII The SL replaceAll Function: Description and Example

Parameter	**Description**
String	The string to have parts replaced.
String	The regular expression used to replace parts of the specified string.
String	The string to replace the matching substrings with.
Return	**Description**
String	The string with matching substrings replaced.
Example	
`var exampleString = “one two three four”;`	
`replaceAll (exampleString, “ ”, “,”);`	

7.3.12 SL Function - split

The `split` function performs the opposite operation as the `join` function. It returns an array from a string that contains the specified regular expression delimiter. The split function is described in Table XIX.

Table XIX The SL split Function: Description and Example

Parameter	**Description**
String	The string to be split.
String	The delimiter to split on.
Return	**Description**
Array	An array of the elements split on the specified delimiter.
Example	
`var exampleString = “one two three four”;`	
`var exampleArray = split (exampleString, “ ”);`	

7.3.13 SL Function - values

The `values` function, described in Table XX, returns the values from the specified associative array.

Table XX The SL values Function: Description and Example

Parameter	Description
Associative Array	The associative array whose list of values is to be returned.
Return	**Description**
Array	The list of values associated with the specified associative array.

Example
`var associativeArray = (0, “Zero”, 1, “One”, 2, “Two”);`
`return join (“\n”, values (associativeArray)) + “\n”;`	

Calling snippet editor functions allows one to access the design tree. The syntax for calling a snippet editor function is given in Figure 9, Production 45. In summary, the syntax is as follows. The entire function call is surrounded by square brackets. Before the snippet editor name a pound sign (#) is placed. The snippet editor name is the value of the first expression, most commonly a literal string (however, it can be a string value from a variable as well). If parameters are needed for the snippet editor function call, a colon is placed in between the snippet editor name and the list of comma separated parameters. Each parameter is an expression. The most common parameters take one of three possible forms, as shown in Figure 11.

```
1. +Snippet Type:Snipplet Code
2. +Snippet Type
3. -Snippet Type
```

Figure 11. Most Common Parameter Formats for Snippet Editor Function Calls

Most types of snippet editors that contain snippets support these parameter formats. These allow one to filter certain types of contained snippets and to call functions on certain types. The leading plus (+) stands for an additive snippet type. The leading minus (-) is for a subtractive type. If the first parameter is additive, the list of acceptable types begins as an empty list. If the first parameter is subtractive, the list of acceptable types begins as a list with all possible types. The specified types include the exact names of snippets and the types that they extend. If a snippet is of type `Assignment`, which extends `Expression` and `Statement`, than `+Expression` would also accept the `Assignment`. To counter this, one could use `+Expression` followed by `-Assignment`.

The default Snipplet code for an accepted type is "return template ();". If one modifies this value, the modification is a substitution. That is, instead of running "`return template ();`" the script would run, for example, "`return asFunctionPrototype ();`". This example case is used for functions because it is often necessary to extract a *function prototype* from a *function implementation*, as it is the case in the programming languages C and C++.

7.4 Special Variables

A few special variables are available to Snipplet scripts. These are summarized in the following subsections.

7.4.1 Special Variables - $rootSnippetGraph and $snippetGraph

These variables give scripts access to the design tree. The `$rootSnippetGraph` variable provides a pointer from the `SnippetGroup` snippet that contains all the code in the project to the root of the design tree. The `$snippetGraph` is a pointer from the current snippet to its location in the design tree. These variables are hash tables. The keys of the hash are the names of the snippet editors contained in the snippet. An additional key, the empty string, is provided to give a reference to the snippet itself. For most types of snippet editors, the value of a hash element will be a linked list of all snippets contained by the snippet editor. The first element of such a list is reserved for the class and the name of the snippet editor (a pair of strings separated by a colon).

7.4.2 Special Variable - $language

The `$language` variable stores the name of the language that is used for code output such as “C”, “C++”, or “Java”.

7.4.3 Special Variable - $projectName

The `$projectName` variable stores the name of the project.

7.4.4 Special Variables - @ and $1, $2, $3,…

These special variables are used for function parameters. The array variable @ stores all the parameters of the array. Each parameter can also be accessed individually using the dollar sign ($) variables. Unlike with PERL, parameters that are complex types such as arrays are not reduced to become part of the parameter array. Rather, the whole array is one element of the parameter array.

Chapter 8 – Writing Programs in Redwood

8.1 Using Existing Snippets

To use an existing snippet, one may simply drag a snippet, listed by name in the `Snippet Chooser` tool, and drop it into 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. First, with some types of snippet editors, editing text and/or manipulating controls will help customize the snippet. Second, with 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, and in a `LineEditor` only a single line of text may be input.

The `Select Build Language` option of the `Project` menu allows a programmer to select the language desired for output. The current version of Redwood does not perform checking for compatibility with already-dropped snippets, but a future version will. The current Beta 2 release 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.

8.2 Creating New Snippets

Since Redwood is a newly developed programming environment, the number of snippets it contains currently is rather limited. Once the online integration is complete, a much wider variety of snippets will be available. Still, from time to time, one may need to develop custom snippets for solving particular problems. The goal of Redwood is, of course, that after one completes a custom snippet, that one could uploaded it to the public snippet server for the general use in the integrated online library.

There are two basic ways to create new snippets. The first is to create a snippet from scratch. The second is to combine other snippets and save the combination as a new type of snippet. Even though the first method is more powerful, it is also slightly more complicated. Section 8.1 has indirectly discussed methods for creating new snippets with the second method. This section discusses creating snippets from scratch. For illustration purposes, this section will cover the creation of a `foreach loop` snippet compatible with the C++ and Java languages, as well as a sigma summation operator.

8.2.1 Foreach Loop Snippet

One of the most useful examples for demonstrating the power of snippets is to show that one can extend a language to include new structures. This fact is already demonstrated with the included snippets for classes that are compatible with C. To create a new snippet, for a structure or any other programming component, one should begin by thinking of mappings to common languages. A `foreach loop` is a loop that goes through each element of an array, assigning the current element to a variable, for convenient access. The programming language PERL makes use of these structures. For languages such as C and C++ it does not necessarily make sense to have such support included. However, when one creates an array class, the usefulness becomes more apparent. For a language such as Java, it would have made sense to allow `foreach loops`. Still, for some reason they were left out of the specification. Fortunately, Redwood's snippets allow one to get around these limitations of the language grammars by creating mappings to those languages. A `foreach loop`'s mapping is simple to imagine. Instead of directly using `foreach loop` syntax, one can use `for loop` syntax in combination with an extra variable that is set for each element. The `for loop` has *n* iterations, where *n* is the length of the array.

The syntax for a PERL `foreach loop` is shown in Figure 12.

```
foreach variable (array)
{
   …
}
```

Figure 12. The Syntax of PERL *foreach* Loops

Using the PERL syntax as a model, one can fairly easily create the snippet display section. The display will consist of four rows. The first row will contain the structure heading "`foreach variable (array)`". The second and fourth rows will contain only the left and right braces, respectively. The third row will contain the loop's code.

To create the first row, the structure heading, one will need a nested table that contains five columns. These columns will hold the "foreach" label, the variable name input (an `IdentifierEditor`), the open parenthesis, the array name input (an `IdentifierEditor`), and the closing parenthesis. The third row will contain a single `CodeEditor` and will have a left margin of ten pixels, to create a tabbed appearance for the code. Figure 13 shows the snippet's entire display section.

```
<display>
   <table>
      <tr>
         <td>
            <table><tr>
               <td>
                  <LineEditor>
                     <param name="Text"><String value="foreach "/></param>
                     <param name="Editable">boolean value="false"/></param>
                  </LineEditor>
               </td>
               <td>
                  <IdentifierEditor name="loopvariable">
                     <param name="Text">String value="variable name"/></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>
                  <IdentifierEditor name="array">
                     <param name="Text">String value="array variable name"/></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>
            </tr></table>
         </td>
      </tr>
```

Figure 13. Snippet Display of foreach Loop

```
      <tr>
         <td>
            <LineEditor>
               <param name="Text"><String value="{"/></param>
               <param name="Editable"><boolean value="false"/></param>
            </LineEditor>
         </td>
      </tr>
      <tr>
         <td width="100%">
         <table margin-left="10"><tr>
               <td width="100%">
                  <CodeEditor name="code">
                     <param name="Allow"><LinkedList>
                        <String value="Statement"/>
                        <String value="Variable Declaration"/>
                     </LinkedList></param>
                     <param name="MinimumSize"><Dimension>
                        <param name="Size"><int value="-1"/><int value="10"/></param>
                     </Dimension></param>
                  </CodeEditor>
               </td>
            </tr></table>
         </td>
      </tr>
      <tr>
         <td>
            <LineEditor>
               <param name="Text"><String value="}"/></param>
               <param name="Editable"><boolean value=”false”/></param>
            </LineEditor>
         </td>
      </tr>
   </table>
</display>
```

Figure 13. Continued from previous page

After defining the display section, the part that will be immediately visible to the user, one must create the templates that map the snippet’s display and contents into the target programming language that the user has selected. Using the Snipplet language reference presented in Chapter 7 for guidance, one should be able to create the template example as shown in Figure 14.

```
<template language="C++">
   var uid = getUID ();

   return "
      int Snippet_ForEach_length_" + uid + " = " + [#"array"] + "-&gt;getLength ();
      for (int Snippet_ForEach_index_" + uid + " = 0; Snippet_ForEach_index_" + uid + "
         &lt; Snippet_ForEach_length_" + uid + "; Snippet_ForEach_index_" + uid + "++)
      {
         " + [#"array" : "getElementType"] + " " + [#"loopvariable"] + " = " + [#"array"]
              + "-&gt;getElementByIndex (Snippet_ForEach_index_" + uid + ");

         " + [#"code"] + "
      }
   ";
</template>

<template language="Java">
   var uid = getUID ();

   return "
      int Snippet_ForEach_length_" + uid + " = " + [#"array"] + ".getLength ();
      for (int Snippet_ForEach_index_" + uid + " = 0; Snippet_ForEach_index_" + uid + " <
         Snippet_ForEach_length_" + uid + "; Snippet_ForEach_index_" + uid + "++)
      {
         " + [#"array" : "getElementType"] + " " + [#"loopvariable"] + " = " + [#"array"]
            + ".getElementByIndex (Snippet_ForEach_index_" + uid + ");

         " + [#"code"] + "
      }
   ";
</template>
```

Figure 14. Snippet Templates (written in the Snipplet language) of foreach Loop, compatible with C++ and Java

8.2.2 Sigma Summation Snippet

Another example that demonstrates the power of Redwood is to create a snippet that allows the “syntax” of a language to extend beyond what is possible with a standard text editor. In Redwood, one can create graphical representations of object and programming constructs or one can simply provide a non-linear layout. The mathematical “sigma” ($\sum$) operator provides a convenient syntax by which mathematicians can sum values. The syntax is non-linear and therefore cannot, in its standard form, be described using a standard text-editor. Figure 15 shows the standard form of sigma notation.

$$\sum_{i=0}^{100} x_i$$

Figure 15. Standard Sigma Notation

Creating a sigma operator is slightly more complicated than creating a `foreach loop`. To create a sigma operation one effectively needs to create a `for loop` that returns a value. This is of course not possible in most languages. On the other hand, one could create a function. However, this would require that one looks up and then passes a fair number of variables to that function. Conveniently, Redwood takes such considerations into account. For any expression, the result is determined in two parts: the part that calculates the value, and the value (or variable that stores the value). A function called `calculateReturnValue` is used for this purpose. All expressions should implement this function.

The `calculateReturnValue` function returns an array of two string elements. The first element is the code used to calculate the value, or the empty string if this is not applicable. The second element is the name of the variable that holds the result of the calculation. In some cases, when it is guaranteed that a literal will be specified (such as with a `Numeric Value` snippet) this may be a direct value instead of a variable name.

To specify the display section of the `Sigma Summation` one should use two columns. The first column is the sigma portion; the second column is the expression to be summed. The first column contains three rows. This means that the `rowspan` attribute will need to be used for the second column. The last row in the first column should contain a nested table with three columns. In all, the snippet will require three

ExpressionEditors, an IdentifierEditor, and two LineEditors (the latter used as labels). The tricky part comes when creating the sigma character. Fortunately, because XML fully supports UNICODE, this becomes trivial. The sigma character is represented in XML by the character encoding “&#x2211;”. The font size of the sigma label is increased to 24 points. Figure 16. shows the complete display section of the sigma summation snippet while Figure 17 contains the code of the snippet’s template section.

```
<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>
```

Figure 16. Snippet Display of Sigma Summation

```
                              </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>
```

Figure 16. Continued from previous page

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

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

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

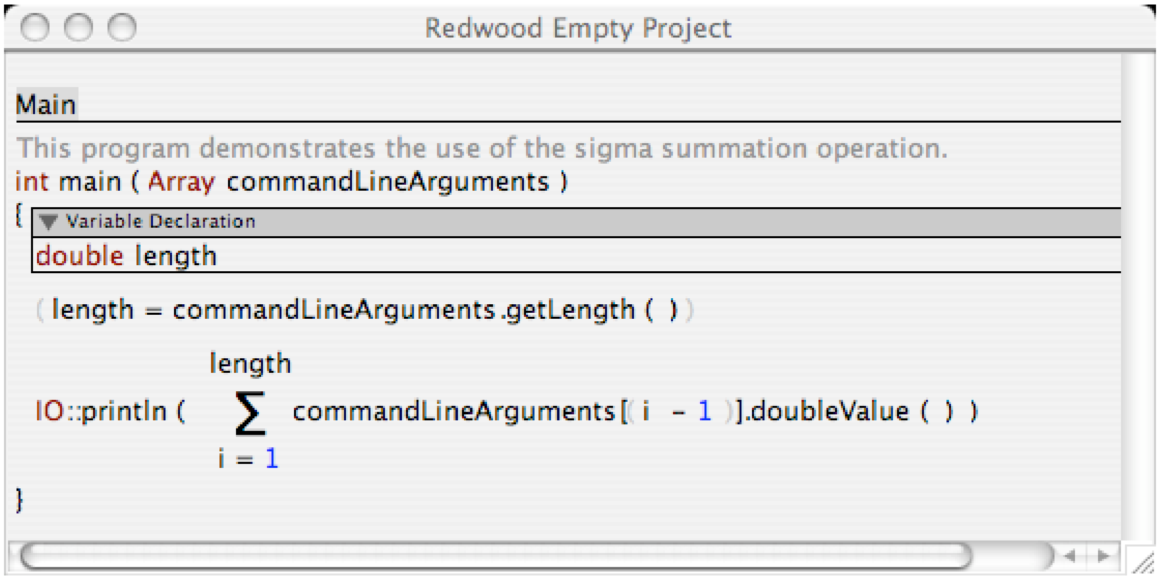
Figure 17. Snippet Templates of Sigma Summation

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

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

Figure 17. Continued from previous page

In Figure 16 one might notice that the `extends` attribute is used in the Snippet tag. The `extends` attribute is used to denote that the snippet is similar to the extended types. Any snippet that extends another snippet should have at least the same, but possibly more, snippet editors (with the same names) and implement at least the same, but possibly more, functions in the templates section (relative to each language) as the extended snippet.

After creating the `Foreach Loop` and `Sigma Summation` snippets, one can use them in Redwood. Figure 18 shows the use of the `Sigma Summation` snippets.



```
This program demonstrates the use of the sigma summation operation.
int main ( Array commandLineArguments )
{
double length
 ( length = commandLineArguments.getLength ( ) )
```

$$\texttt{IO::println (} \sum_{i=1}^{length} \texttt{commandLineArguments[i - 1].doubleValue ( ))}$$

```
}
```

Figure 18. Use of Sigma Summation

Chapter 9 – Conclusions

9.1 Summary of Contributions

Redwood is a complex project that proposes a new type of HPL that is much more adaptable to developer's needs than current systems. In Redwood, snippets provide a powerful method for accessing dynamic parse trees (called design trees), and for dynamically generating source code in various programming languages. In essence, Redwood is a system that allows one to expand the feature set of a programming language beyond what its syntax would naturally support. Furthermore, with Redwood, one can visualize and manipulate programming structures and source code in new and flexible ways. This technology will prove to be beneficial to programmers by allowing them to design, write, and debug code in less time.

Redwood provides a flexible visualization model that allows the proportions of GPL and TPL to be adjusted by the programmer through the use of different snippet sets. Redwood is a general programming tool that is not limited to a specific domain and is scalable for handling large problems.

Through the use of snippets, a programmer is able to use a toolset that is customized to meet the needs for his or her current solutions. The programmer will not be tied to a language that is well suited for one part of a problem but not for another. Further, as innovative TPL technologies are developed, new snippets will be created that use those technologies, allowing the programmer to use the latest tools even when using an "older" target programming language.

9.2 Future Work

Enhancements to Redwood will primarily be directed towards increased portability, usability, and overall functionality.

The most important aspect pertaining to the portability of the Redwood environment is the integration of the online snippets library, as specified in the initial Redwood design document. Support for additional languages and technologies, such as for HTML/JavaScript webpage creation, will also be useful for programmers and designers as Redwood can be used as a general organizational tool. Lastly, support for integrated compilation and execution on non-UNIX operating systems such as Microsoft Windows will be essential to reach the largest possible market.

In terms of usability enhancements, the focus will first be on standardizing the Redwood environment. In particular, support for the clipboard (`cut`, `copy`, and `paste` functions) and `undo`/`redo` stacks are essential for a useful software development package. These features are so useful in fact that they may be added by the time of the public Beta 2 release in Fall 2004. The other priorities will be to create a GUI editor for building snippet displays and possibly simplifying the snippet display syntax for easier editing with a standard text editor. In addition, Redwood needs more configuration options and the ability to save project state. Finally, adding localized versions of Redwood will help broaden the base of programmers that can use the system. Localization is a very important feature that will be added to Redwood. Popular programming languages have historically been English-centric, adding an extra layer of difficulty for non-English speaking programmers. Redwood can help resolve such issues by having different displays (with localized keywords) that map to standard programming structures. Virtually any programmer can create these localized snippets, as knowledge of advanced topics such as compilers will be unnecessary.

Regarding increasing Redwood's overall functionality, a key goal will be to expand the number of included snippets. Of particular interest is the ability to include tools for automatic parallelization, such as `forall loops`. The number of built-in Snipplet script functions should also be increased. Support for a Snipplet preprocessor and advanced syntax error checking will make Snipplet development a much quicker process. In addition, Source code generated by Snipplet scripts will need to be formatted for easier reading, especially for verification purposes (it is likely that a third party product might be used to take care of these needs). Finally, with all of the enhancements that need to be made, it is evident that Redwood could be maintained, in the near future, using the Redwood development environment itself.

References

[1] "Alice: Free, Easy, Interactive 3D Graphics for the WWW", retrieved July 21, 2004 at http://www.alice.org/

[2] Conway, M. J. (1997) "Alice: Easy-to-Learn 3D Scripting for Novices", PhD dissertation, University of Virginia, VA, USA.

[3] KDE HTML Widget, kHTML Library, retrieved October 30, 2003 at http://develhome.kde.org/~danimo/apidocs/khtml/html/index.html

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

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

[6] The Ganssle Group (2000), "Open Source?". retrieved October 30, 2003 at http://www.ganssle.com/articles/opensrc.htm

[7] Apple (2004), "Mac OS X", retrieved August 2, 2004 at http://www.apple.com/macosx/

[8] Scott, M. L. (2000) Programming Language Pragmatics. Academic Press, San Diego, CA.

[9] Wikipedia. "Declarative Programming Language".retrieved July 21, 2004 at http://en.wikipedia.org/wiki/Declarative_programming_language

[10] Norman, D. A. (2002) The Design of Everyday Things. Basic Books, New York.

[11] Pacific Northwest National Laboratory. "A Picture is Worth a Thousand Words", retrieved August 2, 2004, from http://mscf.emsl.pnl.gov/pdf/brochures/visutil.pdf

[12] Sutherland, I. E. (1963) "SKETCHPAD, A Man-Machine Graphical Communication System", Proceedings of the Spring Joint Computer Conference, Spartan Books, Baltimore, MD, p. 329-346.

[13] Smith, D.C. (1975) "PYGMALION: A Creative Programming Environment", PhD dissertation, Stanford University, CA, USA.

[14] Burnett, M. (1999) "Visual Programming". In Encyclopedia of Electrical and Electronics Engineering (J. G. Webster, ed.), John Wiley & Sons Inc., New York.

[15] Visual Basic (2004) Microsoft Corp., Visual Basic Developer Center, retrieved February 14, 2004 from http://msdn.microsoft.com/vbasic/

[16] Cincom Corp., Cincom Smalltalk Homepage (2004) retrieved February, 14, 2004 from http://smalltalk.cincom.com/Home.ssp

[17] Smith, D., Cypher, A., and Spohrer, J., Kidsim: programming agents without a programming language. Communications of the ACM 37(7): 54-67, July 1994.

[18] Byte (1994) "Develop Your Own Automated Phone Applications with PhonePro", retrieved August 2, 2004 from http://www.byte.com/art/9403/sec7/art8.htm

[19] "Advanced Visual Solutions Modeling and Simulation Solutions for Scientific and Real-Time Computing", retrieved August 2, 2004 from http://solutions.intergraph.com/core/avs/

[20] Bishop, R. H. (2001) LabVIEW Student Edition, Prentice Hall, Upper Saddle River, NJ.

[21] Boshernitsan, M. and Downes, M. (2004) "Visual Programming Languages: A Survey", retrieved Feb. 2004 from http://www.cs.berkeley.edu/~maratb/cs263/paper/paper.html

[22] Burnett, M. (2004) "Visual Programming Languages Bibliography", retrieved February 14, 2004 from http://cs.oregonstate.edu/~burnett/vpl.html

[23] Reed, D. A., Aydt, R. A., Madhyastha, T. M., Noe, R. J., Shields, K.A. et al. (1992) "An overview of the Pablo performance analysis environment", Technical report, University of Illinois, Urbana, Illinois.

[24] Babaoglu, O., Alvisi, L., Amoroso, A., Davoli, R. and Giachini, L. A. (1992) "Paralex: An Environment for Parallel Programming in Distributed Systems", Proceedings of the ACM international Conference on Supercomputing, p. 178-187.

[25] Kacsuk, P. et al. (1997) "A Graphical Development and Debugging Environment for Parallel Programs", Parallel Computing, Vol. 22, p. 1747-1770.

[26] Scheidler, C. and Schafers, L. (1993) "TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications", Procs. of PARLE'93: Parallel Architectures and Languages Europe, Munich, Germany, p. 403 – 413.

[27] "CODE Visual Parallel Programming System", retrieved August 2, 2004 from http://www.cs.utexas.edu/users/code/

[28] Tremblay, J. P. “Algorithms”, *Introduction to Computer Science: An Algorithmic Approach*, pp. 18-23, McGraw-Hill, 1989.

[29] Westphal, B. T., Harris, F.C., Jr., Fadali, M. S. “Graphical Programming: A Vehicle for Training Computer Problem Solving.” Proceedings of Frontiers in Education (FIE '03). November 5-8, 2003, Boulder, CO.

[30] Westphal, B. T, Harris, F. C, Jr., Dascalu, S. M. “Snippets: Support for Drag-and-Drop Programming in the Redwood Environment.” Proceedings of the 8th Brazilian Symposium on Programming Languages (SBLP-2004), Niteroi, Rio de Janeiro, Brazil, May 26-28, 2004, pp. 116-127. Also to appear in the Journal of Universal Computer Science, 2004.

[31] “Perl.com: Documentation”, retrieved August 3, 2004 from http://www.perl.com/pub/q/documentation

[32] “Core JavaScript Reference 1.5”, retrieved August 3, 2004 from http://devedge.netscape.com/library/manuals/2000/javascript/1.5/reference/

[33] “Redwood”, retrieved August 3, 2004, from http://www.cs.unr.edu/redwood/

[34] Kew, N. (2000) “CGI Programming FAQ”, retrieved August 3, 2004 from http://www.htmlhelp.com/faq/cgifaq.html