Journal of Universal Computer Science, vol. 10, no. 7 (2004), 859-871
submitted: 16/2/04, accepted: 21/6/04, appeared: 28/7/04 © *J.UCS*

Snippets: Support for Drag-and-Drop Programming in the Redwood Environment

Brian T. Westphal
(University of Nevada, Reno, USA
westphal@cs.unr.edu)

Frederick C. Harris, Jr.
(University of Nevada, Reno, USA
fredh@cs.unr.edu)

Sergiu M. Dascalu
(University of Nevada, Reno, USA
dascalus@cs.unr.edu)

Abstract: This paper presents an overview of the Redwood programming environment and details one of its key features, snippets. Through snippets, developers can both make use of a variety of predefined programming constructs and build their own reusable program components. Language-independent, snippets are descriptions of program parts that can be as simple as an assignment statement or as complex as a sophisticated optimization algorithm. In Redwood, snippets also provide support for a distinguishing facility of visual environments: direct manipulation via drag-and-drop. An example of working with snippets, including snippet definition, visualization, customization, and mapping to code is also presented in the paper.

Key Words: snippets, software development, programming environment, visual programming, design tool, drag-and-drop, Redwood
Categories: D.2.6, D.2.2, D.1.7, D.1.5, D.2.10, H.5.2

1 Introduction

Developed recently in the Computer Science Department of 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 built-in on-line library of snippets, programmers are even able to create new software with little or no code at all!

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 design hierarchy, 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 operational version available currently from the project's website [Redwood 2004].

This project was initially inspired by Carnegie Melon's Alice software package [Alice 2004]. Although Alice is an excellent program for instruction, Redwood has been imagined 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.

Based on the work on Redwood, two papers have been recently published, the first on the subject of using visual programming in computer science education [Westphal, Harris and Fadali 2003], the second describing the design goals and main construction principles of the environment [Westphal, Harris and Dascalu 2004].

In this paper, a brief overview of Redwood is presented and details of one of its most important architectural solutions, snippets, are provided. By making use of snippets, developers can both select, from a toolset, a variety of predefined programming constructs (program components) and build their own reusable components. As detailed later in the paper, snippets are language-independent descriptions of program parts that can be as simple as assignment statements or blocks of code or as complex as versatile optimization or scheduling algorithms. In Redwood, snippets also provide support for a distinguishing facility of visual environments: direct manipulation via drag-and-drop. An example of using snippets, including definition, visual representation, customization, and mapping to code is also presented in the paper.

The remainder of the paper is structured as follows: Section 2 presents a review of Redwood's background literature, Section 3 gives a brief overview of the environment, Section 4 focuses on the snippet concept, Section 5 provides an example of creating and using a new snippet, Section 6 outlines directions of future work, and Section 7 concludes the paper with a summary of Redwood's innovative aspects and potential for further development.

2 Background

The developments of computer graphics first and then of graphical programming systems have changed programming in particular and problem solving in general. The history of this field begins in 1963 when Ivan Sutherland developed Sketchpad [Sutherland 1963], the first interactive computer graphics application. This development opened an entire 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 [Smith 1975], 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 [Burnett 1999]. 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 [Visual Basic 2004] and ParcPlace System's VisualWorks for Smalltalk [Cincom 2004].

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. These approaches vary widely as to the domain they wish to deal with. For example, Stagecoach Software' Cocoa is directed towards graphical simulations and games, Cypress Research's PhonePro handles telephone and voice-mail, Advanced Visual Solutions' software deals with scientific visualization, and National Instruments' LabVIEW focuses on laboratory data acquisition [Bishop 2001].

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 [Boshernitsan and Downes 2004] as well as in Margaret Burnett's Visual Programming Language Bibliography [Burnett 2004]. 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 development systems such as Code from the University of Texas [Newton and Browne 1992] and Pablo from the University of Illinois [Reed et al. 1992] to systems such as Paralex [Babaoglu et al. 1992], Grade [Kacsuk et al. 1997], and Trapper [Scheidler and Schafers 1993]. Many of these systems used similar iconic designs, while others incorporated graphs and connection-based constructs.

At the University of Nevada, Reno, USA we have experimented with several graphical programming systems [Westphal, Harris and Fadali 2003], and have begun the development of our own system. The Redwood Environment [Westphal, Harris and Dascalu 2004] was primarily inspired by the visual programming system Alice, created at Carnegie Melon University by the Stage3 research group [Pausch et al. 1995], [Alice 2004]. However, unlike Alice, Redwood aims to be a more general solution for a larger community of programmers. Alice employs drag-and-drop programming primarily for educational purposes, in particular for teaching programming and computer-based problem solving techniques to students. By contrast, Redwood is designed for the wider community of programmers, from the occasional programmers to the most experienced professional software developers.

3 Overview of the Redwood Environment

The "look and feel" of Redwood can be grasped from [Fig. 1], which shows the environment's user interface. In summary, Redwood's window is divided in two main areas: a *work space* on the left-hand side, where programs are built using a combination of drag-and-drop manipulation of available programming constructs and

text entry customization of these constructs, and a *tools space* on the right-hand side, where support for the actual construction of the program is provided via toolsets such as Favorites, Statement Builder, and Snippet Chooser.

The toolsets on the right hand side of Redwood's window provide access to existing programming constructs (program components), which can be selected and dragged-and-dropped in the workspace. Notably, it is also possible that after a component is customized in the workspace, it can be dragged-and-dropped back in certain areas of the tools space, thus being made available for reuse in other projects.

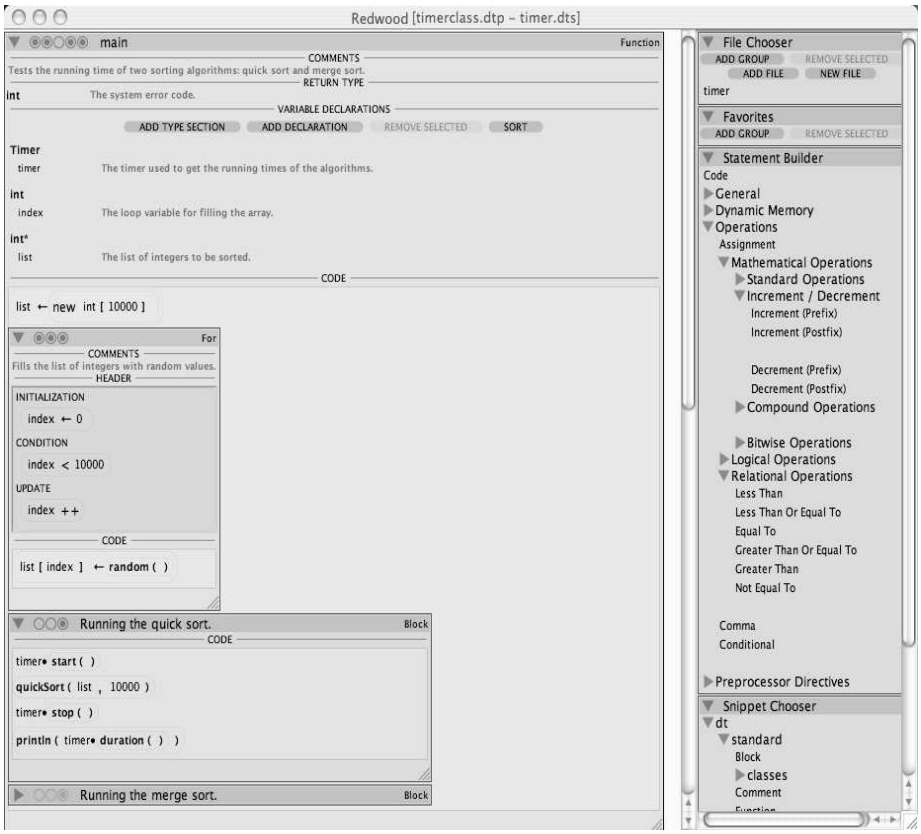

Figure 1: Redwood's Interface

A major abstraction employed in Redwood's design is that of *tree structure*, which can be used to describe any program (in fact, the name of the environment was inspired by this "design tree" abstraction) [Westphal, Harris and Dascalu 2004]. Based on this abstraction, a programmer can move easily between the large-scale representation of a problem's solution and its smaller details. In practical terms, moving between levels of abstraction is made possible through mechanisms such as *snippet nesting* (snippets will be discussed in more details in the next section, for the moment it suffices to say that the main function in [Fig. 1] is a snippet, as it is the for

loop in main's code) and *disclosure triangles and dots* (visible in [Fig. 1] on the top-left corners of the main function, the for loop in main's code, and the running the quick sort block of code, also in main).

As described in the following section, a second major concept that underlies Redwood's design philosophy is that of *snippet*. In [Fig. 2], details of two Redwood toolsets are provided, both built in conjunction with the snippet concept. From both, the developer can select and then drag-and-drop program constructs ("selectable snippets") such as simple assignment operations or more complex class templates.

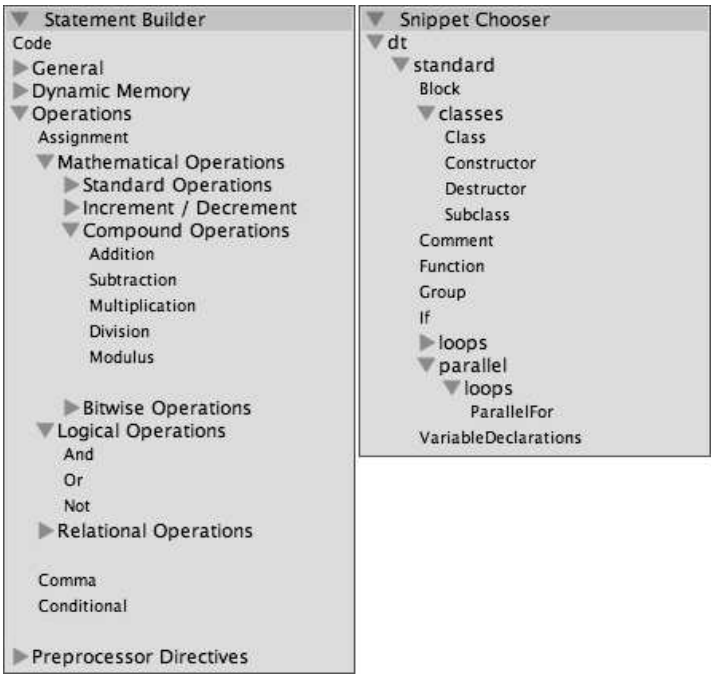

Figure 2: Redwood tool panes: Statement Builder and Snippet Chooser

4 The Snippet Concept

When asked recently "What about the notion of complexity as the primary reason for software bugs? Do you have any concrete ideas on how to reduce complexity?," Victoria Livschitz of Sun Microsystems responded: "I see two principal weapons. One is the intuitiveness of the programming experience from the developer's point of view. Another is the ability to decompose the whole into smaller units and aggregate individual units into a whole" [Heiss 2004]. Using snippets as key tools, the Redwood environment is geared specifically to supply these weapons.

Language-independent, snippets provide a means by which "intuitiveness" can be encapsulated, in the sense that problem-solving solutions become separated from the idiosyncrasies of specific programming languages. In connection with Redwood's design trees, snippets also allow developers to modify and view a program at varying

levels of detail – a "weapon" along the lines of Livschitz's recommended decomposition of the whole into smaller parts and aggregation of individual parts into a whole. A snippet can be reduced to its description, to its content, or to a partial view of its content. By providing support for drag-and-drop programming, pieces of a solution can be treated as "individual units," smaller parts of a larger solution. In fact, a definition for *snippet*, as used in Redwood, is that of a template in which a solution to a smaller part of a problem is embedded.

The Redwood system includes an assortment of snippets, many of them corresponding to traditional programming language constructs such as assignment or decision statements, loops, blocks of code, functions, classes, and so forth. Nevertheless, the true power of snippets comes from the fact they can represent significantly more complex concepts than core programming language constructs. For example, a snippet can be built to embed the solution for a certain type of optimization problem (say, for task allocation in a computer network). In the visual context of Redwood, the encapsulation of an algorithm into a snippet makes possible the manipulation of the algorithm in ways unavailable in traditional, text editing-based software development environments. Representative for Redwood's flexibility and compelling from the user's efficiency standpoint, a programmer can move an "algorithm" around as if it were a physical object, dragging it through the program's workspace and dropping it in that program's design tree node where the developer deems it is best suited for the overall problem's solution.

In Redwood, a snippet is defined and used in a process that can be represented, with certain simplifications, as shown in [Fig. 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 loaded into Redwood's Snippet Chooser toolset, thus the snippet becoming available for use ("selectable," or ready to be dragged-and-dropped). Then, if needed in the program, when the snippet is dragged in 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 saved or the program is compiled and executed, the snippet is mapped onto its associated code according to the templates section of the snippet definition (details are provided on the next page). Note that for simplicity some details of the snippet development stages are omitted in [Fig. 3], as for example a customized snippet can be dropped back in Redwood's tools space, specifically in its Favorites toolset. Thus, a customized snippet can "go back" and become a "selectable snippet," available for reuse.

As shown in [Fig. 4], the XML description of a snippet has two sections, the first being the *display section* (or *component visualization section*) and the second the *templates section* (or *code mapping section*). In the first section the graphical arrangement of intra-snippet components such as expression, comment, editors, and code editors can be defined such that they are presented on screen in a logical fashion.

In the second section, a template describes the output code corresponding to the snippet in a particular language or group of languages. Thus, the source code output is a mapping from the template's parameter values onto the language's programming constructs. Both the display and the template sections are scriptable for creating dynamic interactions.

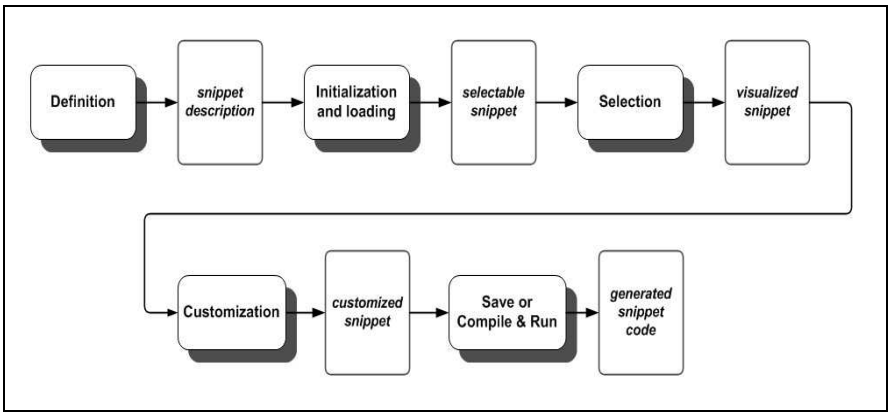

Figure 3: Snippet development stages

```
<DEFINE-SNIPPET NAME="...">
   <COMPONENTS>
      ...
   </COMPONENTS>
   <TEMPLATE LANGUAGES="...">
      ...
   </TEMPLATE>
</DEFINE-SNIPPET>
```

Figure 4: Snippet definition frame

When a snippet is dragged and dropped into Redwood's workspace its XML description is interpreted and graphical components are drawn onto the screen based on the information included in the display section of the description. In the last step of the process shown in [Fig. 3], the parameters and scripts are used to generate source code output for a selected language. More details on creating and using snippets are provided in the following section.

5 Working with Snippets: An Example

As shown in [Fig. 3], the first step in developing a new snippet is to define the snippet using an XML description that has the structure presented in [Fig. 4]. Both the visualization section and the code mapping section of the snippet need to be filled out in order to completely define the snippet. In typical circumstances, creating the visual description of a snippet is straightforward, as one merely needs to describe which

primary components are required. For example, as shown in [Fig. 5], the visual components of a Class snippet include the class description comments, the class name, its lists of attributes, and its methods. Note that, using the previously introduced terminology, this is a base snippet, or generic snippet, as it needs customization for actual use in an executable program. The rather bare visual representation of the generic Class snippet is shown in [Fig. 6].

```
<DEFINE-SNIPPET NAME="Class" BGCOLOR="240, 220, 200" TITLE="name">

  <COMPONENTS>

    <COMPONENT TYPE="CommentEditor" NAME="COMMENTS" TITLE="COMMENTS"/>
    <COMPONENT TYPE="VariableDeclarationEditor" NAME="PUBLICDECLARATIONS"
      TITLE="PUBLIC DATA MEMBERS"/>
    <COMPONENT TYPE="VariableDeclarationEditor" NAME="PROTECTEDDECLARATIONS"
      TITLE="PROTECTED DATA MEMBERS"/>
    <COMPONENT TYPE="VariableDeclarationEditor" NAME="PRIVATEDECLARATIONS"
      TITLE="PRIVATE DATA MEMBERS"/>
    <COMPONENT TYPE="CodeEditor" NAME="PUBLICCODE" TITLE="PUBLIC CODE"/>
    <COMPONENT TYPE="CodeEditor" NAME="PROTECTEDCODE" TITLE="PROTECTED CODE"/>
    <COMPONENT TYPE="CodeEditor" NAME="PRIVATECODE" TITLE="PRIVATE CODE"/>

  </COMPONENTS>

  <PROTOTYPE LANGUAGES="C++">

     class $CLASS.titleForIdentifier;

  </PROTOTYPE>

  <TEMPLATE LANGUAGES="C++">

/**
  $COMMENTS.formatForBlockComment
 */
     class $CLASS.titleForIdentifier
     {
       public:
          $PUBLICDECLARATIONS
       protected:
          $PROTECTEDDECLARATIONS
       private:
          $PRIVATEDECLARATIONS
       public:
          $PUBLICCODE
       protected:
          $PROTECTEDCODE
       private:
          $PRIVATECODE
    };

  </TEMPLATE>
</DEFINE-SNIPPET>
```

Figure 5: Example of snippet description: a generic Class snippet

During the customization stage of the snippet development process, using a generic snippet, such as the Class snippet, an instantiated snippet can be created by adding details through a combination of direct manipulation (drag-and-drop) of constructs and components available in Redwood's tools space and direct editing of text elements such as attribute names, method names, parameter names, and so forth. Note that in [Fig. 7] not all the elements of the Timer snippet are shown as some of the disclosure triangles and dots on the top-left corners of the visual representation of the snippets are turned off. The same [Fig. 7] also provides the occasion for a reference to the other key concept underlying Redwood's design and construction: the design tree, with its concrete application in the nesting of snippets. Specifically, a larger snippet

such as the Timer class is composed of smaller snippets such as Comments, (block of) Code, and If-Then-Else.

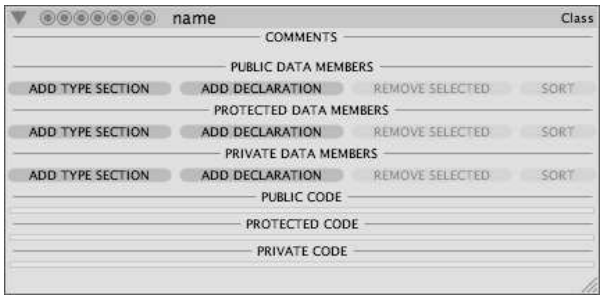

Figure 6: Visual representation of the generic Class snippet

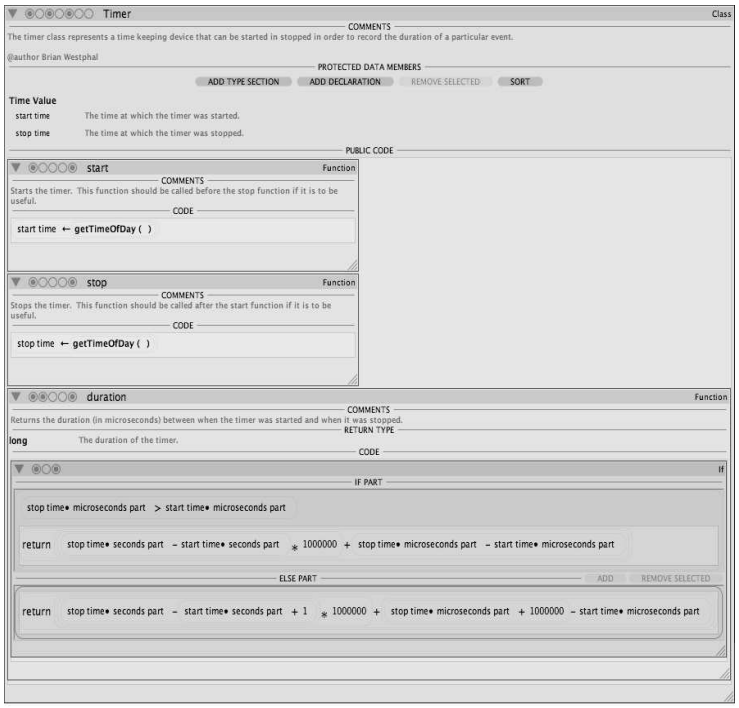

Figure 7: Example of customized snippet: Timer class snippet

The code generated for the Timer snippet when saving the project that contains it (or when compiling the program) is shown in [Fig. 8]. A discussion of directions of future Redwood-related work, including enhancements concerning snippets, follows.

```
class Timer;
/**
 * The timer class represents a time keeping device that can be started in stopped
 * in order to record the duration of a particular event.
 *
 * @author Brian Westphal
 */
class Timer
{
public:

protected:
  /**
   * The time at which the timer was started.
   */
  Time_Value start_time;

  /**
   * The time at which the timer was stopped.
   */
  Time_Value stop_time;

private:

public:

  /**
   * Starts the timer.  This function should be called before the stop function if
   * it is to be useful.
   *
   * @return comments
   */
  void start ()
  {
    (start_time = getTimeOfDay ());
  }

  /**
   * Stops the timer.  This function should be called after the start function if
   * it is to be useful.
   *
   * @return comments
   */
  void stop ()
  {
    (stop_time = getTimeOfDay ());
  }

  /**
   * Returns the duration (in microseconds) between when the timer was started and
   * when it was stopped.
   *
   * @return The duration of the timer.
   */
  long duration ()
  {
    if ((stop_time.microseconds_part > start_time.microseconds_part))
    {
      return (((stop_time.seconds_part - start_time.seconds_part) * 1000000) +
        (stop_time.microseconds_part - start_time.microseconds_part));
    }
    else
    {
      return (((stop_time.seconds_part - (start_time.seconds_part + 1)) * 1000000) +
        ((stop_time.microseconds_part + 1000000) - start_time.microseconds_part));
    }
  }

protected:
private:
};
```

Figure 8: Timer class snippet: generated C++ code

6 Future Work

Several directions are planned for Redwood's future development, among them the following. First, the inclusion of shortcuts and/or automatic translation from typed syntax to Redwood structures is needed to enhance the tool's usability, especially in relation with experienced programmers who are skilled at rapidly typing source code into the computer. Second, support for mapping to additional languages is also needed. Currently, development support is available only for C and C++ programs, but in the near future this support will be extended to languages such as Objective C, Java, Perl, and C#. Third, we are currently considering taking steps towards Redwood's formalization, initially through a model of the environment focused on its key design concepts and functionality. Fourth, our plans include strengthening the current support available in Redwood for parallel programming.

Future work envisaged in relation to snippets is also broad. The first need is for a specialized snippet editor, as it will greatly enhance the ability to modify and create new snippets. Related to this, another direction of further development is concerned with the creation of a library of snippets. This would increase the number and diversity of snippets available, thus serving better the highly desired goal of reusability. As we see it, this library could grow along the expansion of the Redwood community, another important objective of the Redwood project. To achieve this, integrated online support built-in the Redwood environment is also planned.

7 Conclusions

This paper has presented an overview of the Redwood visual programming environment and detailed one of its most important features, snippets. The environment distinguishes itself through its advanced support for visual representation of program structures, direct manipulation of programming constructs, and extended tool support.

Snippets are important to Redwood for a variety of reasons. First, they provide powerful means for visualization and direct manipulation of program components. Second, they offer excellent support for code extensibility and reusability. Third, they are language-independent, which clearly opens the door for Redwood's applicability to a variety of future software projects.

The development of Redwood is being intensively pursued by our team at the University of Nevada, Reno. A group of faculty and students is currently working on the design and implementation of the integrated on-line support mentioned above as well as on some of the other Redwood-related enhancements. This work is proceeding at a fast pace because we believe the environment has high potential in terms of enhancing programmers' efficiency and accuracy.

References

[Alice 2004] "Alice: Free, Easy, Interactive 3D Graphics for the WWW" (2004); http://www.alice.org/

[Babaoglu et al. 1992] Babaoglu, O., Alvisi, L., Amoroso, A., Davoli, R., Giachini, L. A.: "Paralex: An Environment for Parallel Programming in Distributed Systems"; Proc. ACM International Conference on Supercomputing (1992), 178-187.

[Bishop 2001] Bishop, R. H.: "LabVIEW Student Edition"; Prentice Hall / Upper Saddle River, NJ (2001).

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

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

[Burnett 2004] Burnett, M.: "Visual Programming Languages Bibliography" (2004); http://www.cs.oregonstate.edu/~burnett/vpl.html

[Cincom 2004] "Cincom Smalltalk Homepage"(2004), http://smalltalk.cincom.com/Home.ssp

[Heiss 2004] Heiss, J.J.: "The Next Move in Programming: A Conversation with Sun's Victoria Livschitz" (2004); http://www.java.sun.com/developer/technicalArticles/Interviews/livschitz_qa.html

[Kacsuk et al. 1997] Kacsuk, P., Cunha, J.C., Dosza, G., Lourenco, J., Fadgyas, T., Antao, T.: "A Graphical Development and Debugging Environment for Parallel Programs"; Parallel Computing, 22 (1997), 1747-1770.

[Newton and Brown 1992] Newton, P., Browne, J.C.: "The CODE 2.0 Graphical Parallel Programming Language", Proc. ACM International Conference on Supercomputing (1992), 167-177.

[Pausch et al. 1995] Pausch, R., Burnette, T., Capehart, A.C., Conway, M., Cosgrove, D., DeLine, R., Durbin, J., Gossweiler, R., Koga, S., White, J.: "A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality"; IEEE Computer Graphics and Applications (1995).

[Redwood 2004] "Redwood v1.0 Beta 1" (2004); http://www.cs.unr.edu/redwood/

[Reed et al. 1992] Reed, D. A., Aydt, R. A., Madhyastha, T. M., Noe, R. J., Shields, K.A. et al.: "An Overview of the Pablo Performance Analysis Environment", Technical Report, University of Illinois, Urbana, Illinois (1992).

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

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

[Sutherland 1963] Sutherland, I. E.: "SKETCHPAD, A Man-Machine Graphical Communication System"; Proc. Spring Joint Computer Conference, Spartan Books, Baltimore, MD (1963), 329-346.

[Visual Basic 2004] "Visual Basic Developer Center" (2004); http://www.msdn.microsoft.com/vbasic/

[Westphal, Harris, and Fadali 2003] Westphal, B.T., Harris, Jr., F.C., Fadali, M.S.: "Graphical Programming: A Vehicle for Teaching Computer Problem Solving"; Proc. 33rd ASEE/IEEE Frontiers in Education Conference, Boulder, CO, USA (2003), F4C_19-23.

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