Designing and Explaining Programs with a Literate Pseudocode

GEOFFREY G ROY
Edith Cowan University, Perth

Designing and explaining programs are often difficult tasks, especially when novices are involved. It also concerns more experienced programmers when complex algorithms need to be carefully explained and documented as part of software development.
Good practice suggests that code and documentation be tightly coupled; but there are only a few support tools specifically targeted at novices that require this as an integral part of the programming process. We propose a new tool design with this objective in mind: P-Coder gives a flexible descriptive view of the program via pseudocode notation within a graphical tree-structured model, from which the complete algorithm can be specified to retain the informal description integrated with the final code. Thus the code meets some of the key requirements for realizing a literate program.

Categories and Subject Descriptors: D.1.7 [**Programming Techniques**]: Visual Programming; D.2.3 [**Software Engineering**]: Coding Tools and Techniques --*structured programming*, *top-down programming*

General Terms: Algorithms, Design, DocumentationAdditional Key Words and Phrases: Pseudocode, literate programming, novices

1. INTRODUCTION

While program design is often treated as more of an art than a science, there is much evidence to support the view that a well-organized approach to both design and implementation has considerable benefits, including

- more efficient algorithm development;
- reduction in coding errors;
- more readable code, especially for nonauthor readers; and
- better management of complex systems for integration, maintenance, and support.

CASE tools are meant to achieve these outcomes, but are usually targeted at complex systems where the payoffs are more obvious. CASE tools are often complex themselves, hence there is a steep learning curve in learning how to use them effectively; they are mainly targeted at skilled programmers.

Our interest is primarily focused on novices, perhaps on those in the early stages of their development, where the most basic computational principles are learned (i.e., sequence, iteration, selection, and recursion). We are also interested in more experienced programmers, where some particularly difficult project, perhaps a new algorithm, is being

Author's address: G.G. Roy, School of Computing and Information Science, Edith Cowan University, Perth 6050, Australia; email: g.roy@ecu.edu.au

developed and requires additional design support. The goal is to clarify the computational process and provide support for designing, documenting, and explaining the program.

Some example applications of a new programming tool targeted at these domains are presented here. P-Coder was developed with the specific intent of exposing the structure of a program, so that the computational processes can be understood more easily than by just examining the source code. It is based on the following assumptions:

- pseudocode provides an effective vehicle for describing computational processes;
- a combination of text and graphics allows the pseudocode to be defined, edited, and, when compared to raw code, displayed with increased readability;
- through a process of stepwise refinement the actual code can be built progressively, until the complete program is fully specified and operational.
- by tightly coupling the pseudocode with comments and code segments,
- the resulting code can reach the level of literate programming.

These goals should also apply to any good CASE tool, but our aim is to build a one suitable for novices, where the focus is on the most basic computational principles and less on large-scale and complex software systems.

2. PSEUDOCODE CONCEPTS

Pseudocode aims to fill the gap between the informal (spoken or written) description of the programming task and the final program (code) that can be executed (or at least automatically converted into an executable form). Pseudocode generally includes the following:

- the use of English-like statements (or whatever language is appropriate) to describe the computational task and/or process;
- some reserved words or symbols (nouns and verbs) to describe common processes and actions;
- ways to describe(e.g., phrases) standard computational tasks; and often
- some graphical notation to add clarity and richness to the descriptions.

Pseudocode can be built on either, or a combination of, textual and graphical elements, and many text-based pseudocode variants have been proposed. Some syntactical elements are generally required to represent the four basic computational processes of sequence, iteration, selection, and recursion. In addition, pseudocode notation can include elements to describe input/output operations, file operations, and some level of modularization through function and method/procedure definitions to allow for the reuse of code and for information hiding. In specialized domains there may also be notation to support database or communication operations where they are central to the application/domain being specified.

The evolution of pseudocode has its roots in the early analysis of teaching programming skills to novices. For example, Spohrer et al. [1985] were concerned with studying how novices approached programming tasks and why so many mistakes (bugs) were generated. They proposed a goal and plan tree approach, where the programming tasks (expressed as a set of requirements) were progressively broken down into goals; alternative plans were described to achieve the goals; the plans led to the identification of

subgoals, and so on. In many ways this process is close to the stepwise refinement that is commonly seen today, as described by Reynolds et al. [1992].

Scanlin [1988] studied the effectiveness of text-based pseudocode and graphical flowcharts in helping users to understand both data structures and algorithms. He found that the graphical flowchart provided a clear benefit to the student reader because textual pseudocode is mainly processed by the brain's left hemisphere (verbal, logical, sequential), while the flowchart can also effectively utilize the right hemisphere (visual, spatial, simultaneous) at the same time. The flowchart, with both text and graphic notation, can thus make more effective use of brainpower.

A variety of graphical pseudocode forms have also been proposed (a selection is described by Cross and Sheppard [1988]), including ANSI flowcharts, Nassi-Shneiderman diagrams [Nassi and Shneiderman 1973], Warnier-Orr diagrams [Orr 1977], action diagrams [Martin and McLure 1985], and control structure diagrams [Cross 1986]. The goal of each is essentially the same – to provide a clear picture of the structure and semantics of the program through a combination of graphical constructions and some additional textual annotations. Each style has its strengths and weaknesses in terms of clarity, expressiveness, and (most importantly) the overheads in using the methodologies in practice.

To allow the pseudocode description to grow and evolve effectively, it should be supported by an appropriate tool, without which there is a tedious level of rewriting, adding, and deleting. A number of support tools have been proposed, some examples follow:

The First Programming Language (FPL) as proposed by Taylor et al. [1986], made one of the first attempts at a tool-based graphical programming environment intended for use with the Pascal language. FPL offered a number of functional elements (as icons) that represent the basic computational primitives (sequence, iteration and selection), plus some specialized icons for different kinds of assignments, Pascal block structures, and variable declarations. The goal was to produce executable code directly from FPL; the complete package was presented as an interactive computer-based tool.

SchemaCode [Robillard 1986] offers a more formal tree-structured model of a program, with the aim of generating executable code. It relies on the user inserting some (or all) program structures to match the target language, as well as the full syntax of the required computation.

Starting from the top (root) of the tree model, there is a sequence of nodes as we move down the diagram. Branches are added to define and /or refine computational steps, which can be opened or closed to explore their contents.

The program can be developed incrementally by stepwise refinement, with the leaf nodes (ultimately) containing actual code (in the target language). The computational primitives are shown in various graphic notations, which increase the readability of the diagram that would otherwise appear in the form of the program's actual code.

B-liner [Varatek Software 1999] is based on the Warnier-Orr diagramming model [Orr 1980; Warnier 1976; Escalona 1984] where the concepts and relations are organized into an hierarchical tree using bracket notation. The primary task is defined at the root of the tree (left-most node), and the branches show the increasing detail within each child bracket. The diagram extends both vertically and horizontally.

The algorithm development proceeds top-down, adding detail as the refinement proceeds. A "bracket" contains all the child nodes on that branch of the tree. Within each bracket there is an implied (generally) top-to-bottom sequence, so the first instruction is at the top of the branch and the last instruction at the bottom. It is possible to collapse or expand each branch of the tree to aid in its readability as the size the tree grows.

B-liner provides a tool to describe algorithms in varying levels of detail; it is primarily a tool for algorithm specification. It supports stepwise refinement, but there is no way of specifying, or generating, executable code in the model. B-liner has a range of other computational capabilities relating to "hierarchical spreadsheets" which are built on top of the tree structure (but these are not of interest here).

Grasp [Cross and Barowski 2002] is not really a pseudocode tool, although it does make some useful contributions. In Grasp, the starting point is the actual (syntactically correct) code. It is based on the concept of the control structure diagram (CSD); see Cross and Sheppard [1988] and Cross et al. [1998].

The CSD diagram is created from code; its main purpose is to provide a graphical (and thus more readable) display of the program structure (and the underlying algorithm) after the code is written. The model is changed by editing the code, and the CSD changes are then available for view.

Grasp is in fact a "full" development environment, and has a range of other capabilities (editing, compiling, debugging, and executing), which are not discussed here. Grasp can also generate UML-styled class diagrams.

UML-based tools. In more recent times, and with the development of O-O paradigms, a range of new diagramming techniques have emerged, which are generally integrated into CASE tools. They provide ways of viewing the underlying model within the O-O framework. For example:

- *class diagrams* show the static structure and relationships between classes;
- *state diagrams* show the permissible states and transitions that can occur between states;
- *sequence diagrams* show the temporal dependencies between the different actions that form the basis of the Unified Modelling Language (UML, see Object Management Group: http://www.omg.org/uml/). Each diagram is built from the same underlying computational model and thus highlight, or explain, different views of the same model. UML, however, is probably too complex for novices, although some parts of it are particularly useful in the early, introductory, sages of O-O (e.g., class diagrams).

P-Coder builds from the basic computational primitives of sequence, iteration, selection, and recursion; Figure 1 shows their implementation in P-Coder, displayed as simple tree abstractions. As shown, computations take place within a method node. The order of computation proceeds down the tree, taking each branch to the right until a leaf node is reached. The next computational step is found by backtracking from the leaf node and taking the next available downward path until the complete computational tree has been traversed, or all available pathways are explored.

The icons used for each node are intended to portray something of the semantics, while the associated text provides a way of clarifying the semantics for the reader.

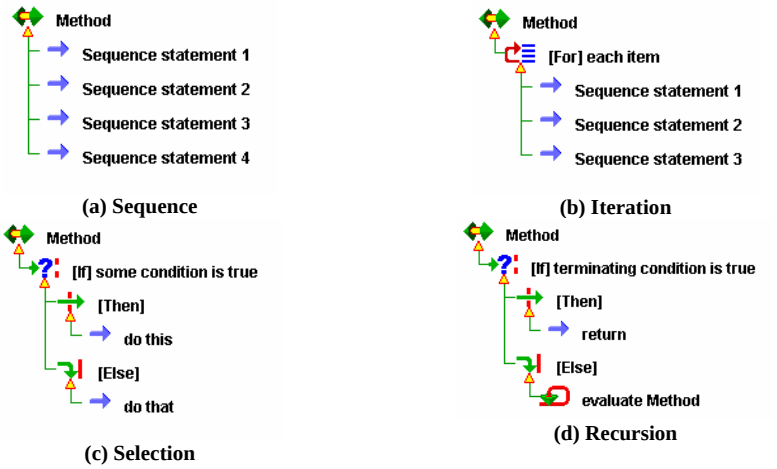

Fig. 1. The P-Coder computational primitives.

At a higher level, P-Coder provides package and class structures in a Java-like fashion, as shown in Fig. 2.

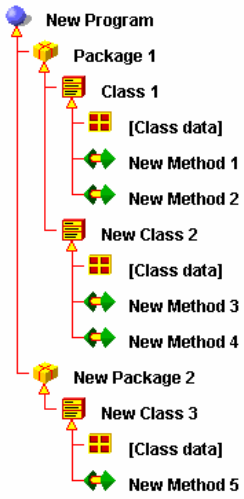

Fig. 2. The package/class structure in P-Coder.

A program is to be composed as a set of packages, which are in turn composed of sets of classes. Each class will contain a class data node to represent class data declarations, and zero or more Methods to describe the functional capabilities of the Class. A tree structure is used here as well with the proviso that there is no significance in the ordering of the nodes (Method nodes and higher) in the tree. As a result a complete program can be built using the same graphical model.

This tree abstraction can be used to represent most computational processes and O-O language constructs, which in general is sufficient for novices and adequate for a wide range of applications, with the exception of the more esoteric language elements. Currently, P-Coder has been implemented for Java.

3. UNDERSTANDING PROGRAMS

While algorithm and program design is central for our project, the readability of a program must also play a central role if the program is to be understood– especially if someone else wrote it originally. For novices, many learning processes are derived from examples that can be copied and modified to suit new contexts, thus encouraging the development of skills at abstraction.

A comprehensive review of research in program readability is provided by Deimel and Neveda [1990], which contains an annotated bibliography of earlier studies relating to the organization and layout of code to improve its readability. The focus tends to be on reading code that has already been created (or code written by others), rather than on reading one's own code as an integral part of a design process. Even so, many of the issues are likely to be similar. Deimel and Nevada give some 30 tips on creating code for better readability; the most relevant for this context are the following:

- inconsistencies between comments and code;
- use of indentation to show structure;
- use of stepwise abstraction; and
- use of program slicing to isolate behavioral components.

This last point is particularly interesting in that it seems analogous to the modularization and encapsulation concepts that have appeared more recently. That is, limiting the complexity of program elements to single functional units (with the use of methods, procedures, functions, etc), and trying to ensure that the scope of variables (and their visibility to the reader) is limited to those parts of the program where they are required.

Deimel and Neveda [1990] are particularly interested in teaching programmers (and novices) how to read programs, and argue that the skills to do so need to be taught. Their focus is on reading the code itself – hence the focus on the stylistic issues surrounding the presentation of code.

To demonstrate program comprehension using the P-Coder notation, we can look at the problem of discovering the roots of the quadratic expression

$$ax^2 + bx + c = 0$$

that are given by

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The algorithm for this problem is more complex than first appears, especially if all possible values of the coefficients *a*, *b*, and *c* are catered for. A top-down approach to explore (and display) the algorithm is shown in five steps, from Figures 3 to 7.

The P-Coder tool allows the program tree to be rolled up or rolled out, as required, at any selected node. This capability, by progressively rolling the pseudocode diagram up and out, can be used to explain how the example program works in five steps:

Step (1) the basic structure of the program is defined (Fig. 3);
Step (2) test for valid data, the precondition for the method (Fig. 4);
Step (3) if the *a* coefficient is zero, then a simple linear expression is solved (Fig. 5);
Step (4) test the sign of the term under square root for imaginary or real roots (Fig. 6);
Step (5) test for the zero term under square root for one or two real solutions (Fig. 7).

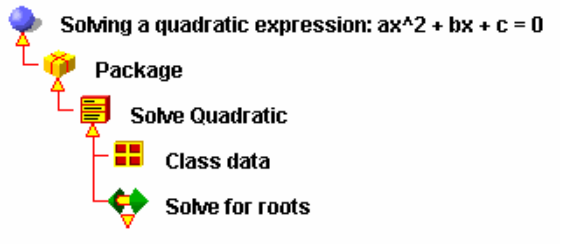

Fig. 3. Quadratic roots - Step (1).

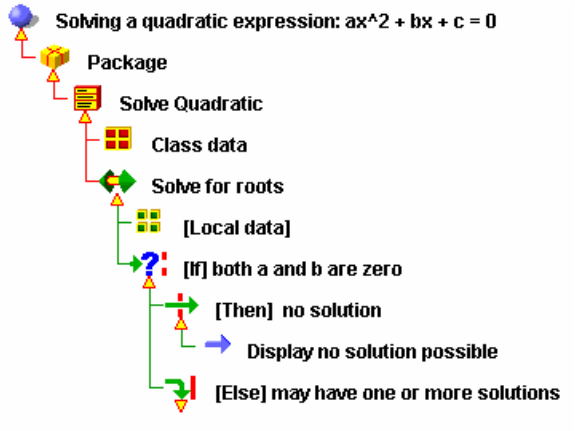

Fig. 4. Quadratic roots - Step (2).

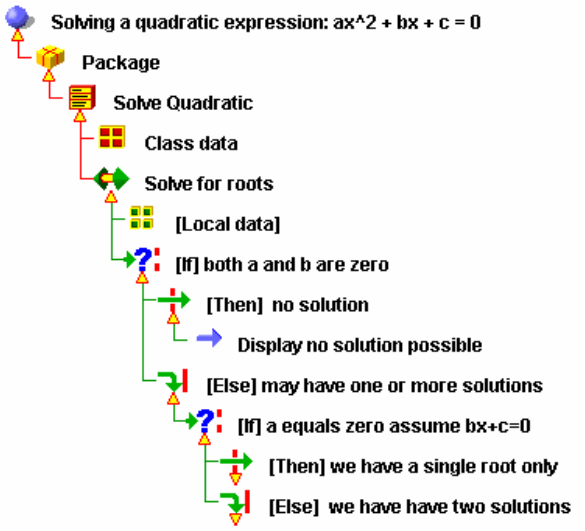

Fig. 5. Quadratic roots - Step (3).

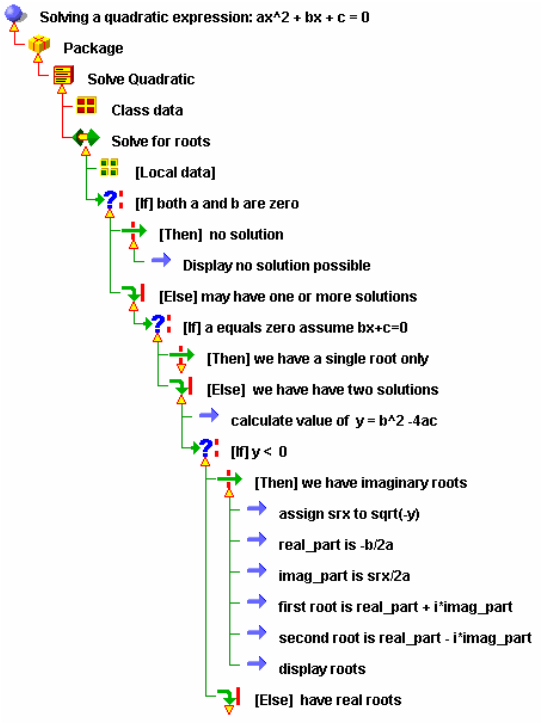

Fig. 6. Quadratic roots - Step (4).

Further expansions of the program are possible to ultimately show all the required computational steps. The creation of the program could follow a very similar process.

The representation of the program in this way clearly describes the algorithm and the computational steps. To achieve this, a number of presentation vehicles are used, including:

- a hierarchical tree structure to describe the computational steps;
- ability to display parts of the model by rolling the model up and out on a node-by-node basis;
- a primary computational sequence that is down and to the right, to follow all available branches that are shown with an indentation style to show processes and subprocesses; and
- both graphical and textual clues used to qualify and enhance the semantics for basic computational processes and underlying algorithms.

4. LITERATE PROGRAMMING

Literate programming concepts [Knuth 1984; Shum and Cook 2002] focus on integrating informal descriptions of the program's functions (perhaps the documentation) with the formality of programming language instructions. A comprehensive review of literate programming can be found at http://www.literateprogramming.com. While the inclusion of comments in code is generally accepted as good practice, there are few tools that require, or support, this in a tightly coupled and formal way. As Thimbleby [2003]

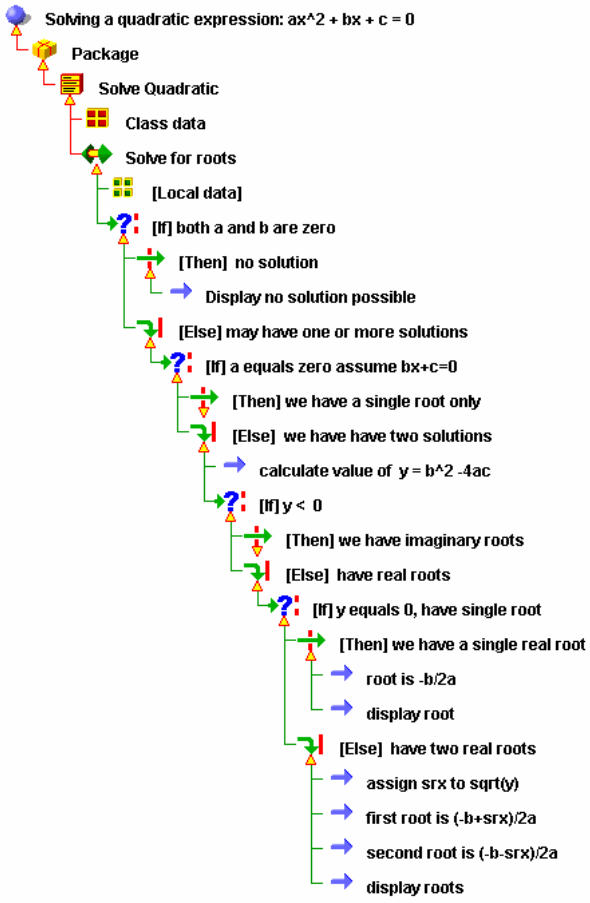

Fig. 7. Quadratic roots - Step (5).

demonstrates, there are a number of tools that support literate programming at various levels and in different environments. The Noweb tool [Ramsey 2003] follows Knuth's original ideas quite closely and appears to have some level of acceptance, though it does presume the use of LATEX as a documentation vehicle.

Knuth's approach is based on the premise that the documentation and code should reside in a "web" on which two processing operations can be performed by *weaving* to produce the documentation and *tangling*: to produce the code ready for compilation and execution. These processes are transformations of a single model (or repository) where all aspects of the model, algorithm, and code are contained.

Thimbleby outlines a number of desirable properties that a literate program should exhibit, the key ones are (in summary):

- documentation and code should develop together and be tightly coupled;
- editing must be possible without affecting the integrity of the documentation and code;
- tool support must be lightweight, easy to use, and discourage manual "touch ups";

- the tool must scale;
- fragments of code should be explainable in any order;
- readers of the documentation should not have to face special notation or conventions;
- the tool should be language independent;
- any required translations from documentation to code should be automated; and
- the tools must be simple.

The concept of literate programming, as proposed by Knuth, aims to produce a typeset document from the weaving process. P-Coder does not go that far, and as a result typeset-ready documentation is not produced, but well-documented code consistent with the pseudocode model, and vice versa, is produced. Most, but not all, of Thimbleby's desirable properties are achieved (the tool may not scale and does not permit the flexible ordering of explanations).

Fig. 8 shows a simple algorithm description (to sum the elements of an array) in P-Coder pseudocode notation. The computational process is represented in tree-structured syntax to describe the algorithm. Each node in the tree represents a computational primitive with a suitable icon and textual note to suggest the associated semantics.

From this pseudocode model it is possible, with some stylistic assumptions, to create the complete structure of the code as shown in Fig. 9, where the actual code is faded out to highlight the generated key words and program structure. The pseudocode notes and most of the syntax elements are also in place. The missing code segments form a small part of the whole program (this is quite typical).

Once the structure of the program is in place, the user can begin adding code. Each node in the model has an associated details dialog (shown in Fig. 10), which is opened by simply <shift-left-clicking> on a node, which is then highlighted to show the dialog that is currently open. Each details dialog is specialized for its node type; hence the user is prompted for the type of information required. Sequence nodes allow general code statements to be entered, as shown in Fig. 11. These dialogs provide a node-specific vehicle for entering the executable components of code for each pseudocode node, and so provide some guidance for the novice user.

With the additional code segments defined, the complete program is contained in the P-Coder model and the executable program can be generated.

Fig. 12(a) shows the complete "literate" code, including the pseudocode notes as well as the explicitly declared comments and the code segments as defined for each node. If required, the pseudocode notes can be suppressed, as shown in Fig. 12(b). Both versions of the code are complete and correct Java program elements.

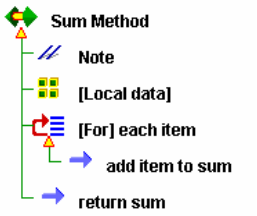

Fig. 8. Simple pseudocode model to sum an array.

```
//=======================================================
//  P-Coder Java Code: Method Only.java
//  Created on: 30 April 2003 7:39:45
//=======================================================

    /**< Sum Method >**/
    // -------------------------------------------------
    // Sum Method
    // -------------------------------------------------
    public int Sum(int [] item)
    {
        /**< Note >**/
        // Computes the sum of an array
        // args - an array of int values
        // returns - the sum of the items in the array
        /**< [Local data]  >**/
        /**< create and initialize sum >**/
        int sum = 0;
        /**< [For] each item in array >**/
        for(int i = 0; i < item.length; i++)
        {
            /**< add item to sum >**/
            sum = sum + item[i];
        }
        /**< return sum >**/
        return sum;
    }
```

Fig. 9. The code structure generated from the pseudocode model.

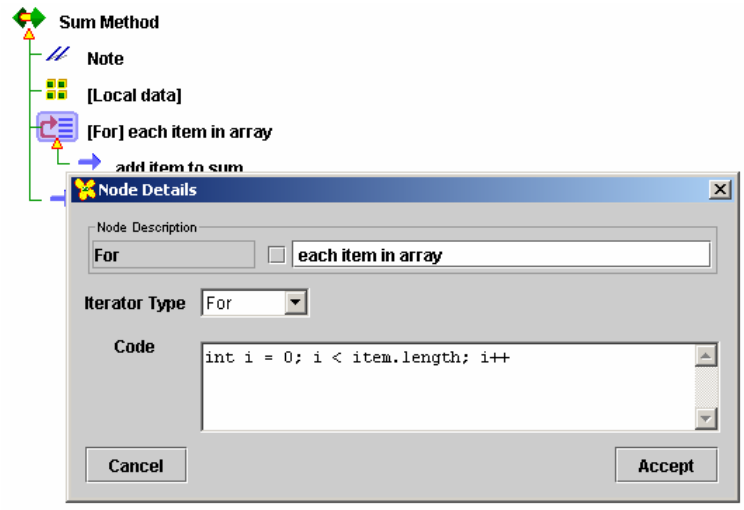

Fig. 10. Opening a node details dialog.

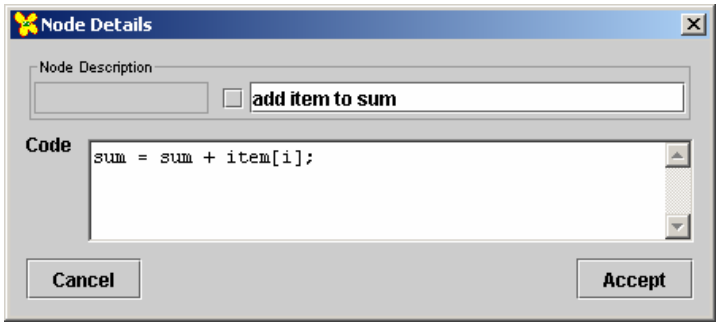

Fig. 11. Example node dialogs.

The structure of the code is generated from the pseudocode model. The annotations are tightly coupled with the code and cannot be separated. As a result, the final code can (and should) contain a substantial body of documentation for the program. Additional notes and comments are accommodated inside the comment nodes that can be added anywhere in the model tree.

5. P-CODER IMPLEMENTATION

It is our proposition that a formal pseudocode representation with integrated text and graphics can be used effectively to describe quite complex algorithms. Algorithms can be explained more clearly than by using either informal text only or the more cryptic syntax of a programming language. The P-Coder tool has been implemented to support this concept.

Fig. 13 presents the P-Coder user interface showing the classic Towers of Hanoi problem. The computational process is quite clear, enabling the semantics to be described fully and clearly. The user interface supports a fully interactive graphic editor for the creation and editing of algorithms and complete programs.

P-Coder supports progressive refinement of the design process with a mainly top-down approach. It is possible to define program elements from the bottom-up, as well as to import program elements (subtrees) from libraries of P-Coder models.

The complete code for the Towers of Hanoi algorithm is shown in Fig. 14. This code, together with the pseudocode, provides an effective definition and description of the algorithm. Since both views are derived from the same model, some of the key requirements for a literate program are achieved.

The P-Coder Designer View (shown in Fig. 13) provides full interactive editing capabilities, including

- the selection of required node types from the icon menu (on the left) and attachment to the model tree;
- nodes that represent computational primitives (sequence, iteration, selection, and recursion), plus others to provide Java-specific language elements (packages, classes, methods, class and local data, and exception handling);
- additional comment nodes that can be inserted anywhere in the model to provide supplementary documentation for the program;

```
//=====================================================
//  P-Coder Java Code: Method Only.java
//  Created on: 30 April 2003 7:43:27
//=====================================================

    /**< Sum Method >**/
    // ---------------------------------------------
    // Sum Method
    // ---------------------------------------------
    public int Sum(int [] item)
    {
        /**< Note >**/
        // -----------------------------------------
        // Computes the sum of an array
        // args - an array of int values
        // returns - the sum of the items in the array
        // -----------------------------------------
        /**< [Local data]  >**/
        /**< create and initialize sum >**/
        int sum = 0;
        /**< [For] each item in array >**/
        for(int i = 0; i < item.length; i++)
        {
            /**< add item to sum >**/
            sum = sum + item[i];
        }
        /**< return sum >**/
        return sum;
    }
```

(a)

```
//=====================================================
//  P-Coder Java Code: Method Only.java
//  Created on: 30 April 2003 7:44:11
//=====================================================

    // ---------------------------------------------
    // Sum Method
    // ---------------------------------------------
    public int Sum(int [] item)
    {
        // -----------------------------------------
        // Computes the sum of an array
        // args - an array of int values
        // returns - the sum of the items in the array
        // -----------------------------------------
        int sum = 0;
        for(int i = 0; i < item.length; i++)
        {
            sum = sum + item[i];
        }
        return sum;
    }
```

(b)

Fig. 12. Example of generated code: (a) complete with pseudocode; (b) condensed without pseudocode.

- a context-sensitive rule-based system to ensure that nodes can only be placed at valid locations as determined by the P-Coder language grammar, which guarantees that invalid computational constructs cannot be formed (but no claims can be made about the code segments inserted into the node dialogs);
- an implementation of a subset of the Java language; some of the more esoteric elements are not implemented (e.g., inner classes);
- a model that can be rolled-up or out at each node to simplify the view and to retain focus on selected parts of the model;

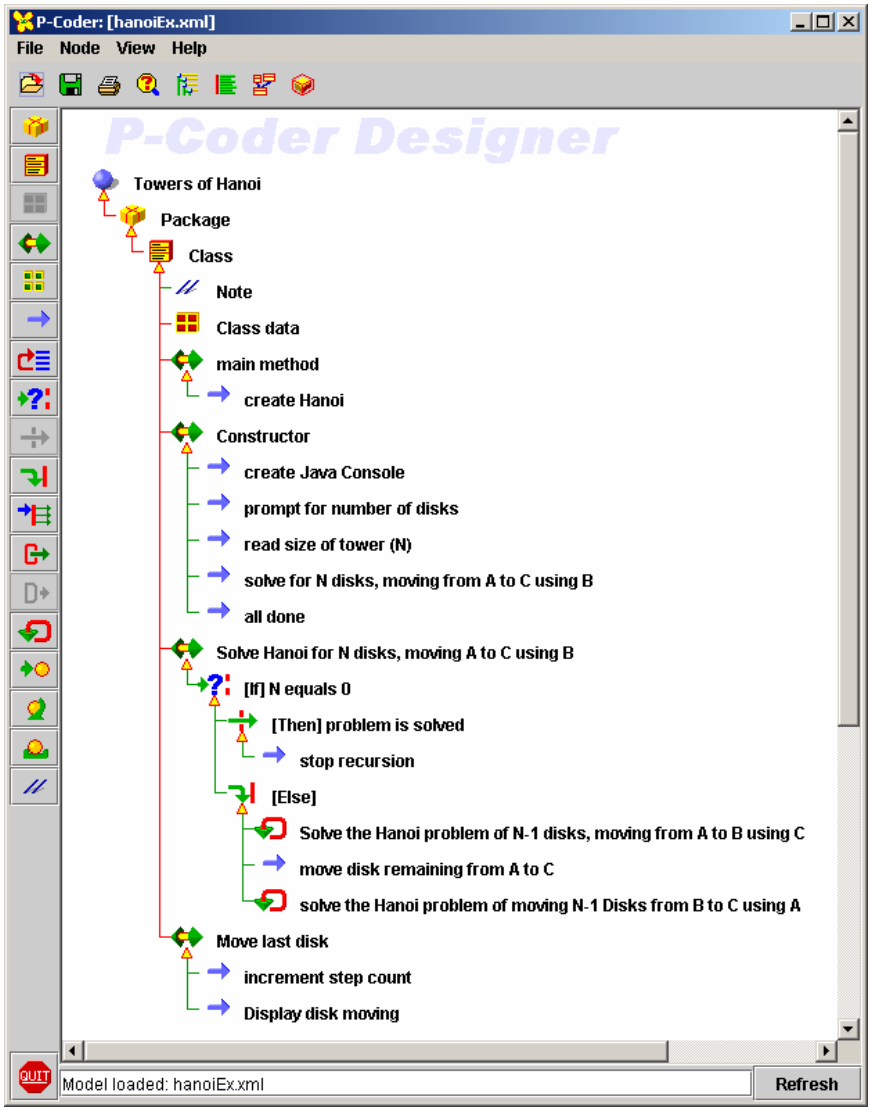

Fig. 13. The Towers of Hanoi algorithm expressed in P-Coder.

```
//============================================================================
//  P-Coder Java Code: Hanoi.java
//  Created on: 31 March 2003 12:21:05
//============================================================================
/**< Package >**/

/**< Class >**/
public class Hanoi
{
    /**< Note >**/
    // The Towers of Hanoi is a classic computational problem
    // that is often used to teach recursion.  The algorithm
    // is interesting in that it solves the problem using
    // two recursive steps, plus a intervening step to
    // move the largest disk to its new, and final, place.
    //
    // The three pegs are labeled A, B and C.
    /**< Class data >**/
    /**< the I/O widow >**/
    JavaConsole con;
    /**< to count the number of teps required >**/
    int step = 0;

    /**< main method >**/
    // ------------------------------------------------------------------
    // main method
    // ------------------------------------------------------------------
    public static void main(String [] args)
    {
        /**< create Hanoi >**/
        Hanoi hanoi = new Hanoi();
    }

    /**< Constructor >**/
    // ------------------------------------------------------------------
    // Constructor
    // ------------------------------------------------------------------
    public  Hanoi()
    {
        /**< create Java Console >**/
        con = new JavaConsole();
        /**< prompt for number of disks >**/
        con.write("Input no of disks:");
        /**< read size of tower (N) >**/
        int N = con.read_int();
        /**< solve for N disks, moving from A to C using B >**/
        solveHanoi(N,"pegA","pegB","pegC");
        /**< all done >**/
        con.writeln("Completed...");
    }

    /**< Solve Hanoi for N disks, moving A to C using B >**/
    // ------------------------------------------------------------------
    // Solve Hanoi for N disks, moving A to C using B
    // ------------------------------------------------------------------
    public void solveHanoi(int N, String A, String B, String C)
    {
        /**< [If] N equals 0  >**/
        if(N == 0)
        /**< [Then] problem is solved >**/
        {
            /**< stop recursion >**/
```

```
                return;
            }
            /**< [Else] >**/
            else
            {
                /**< Solve the Hanoi problem of N-1 disks, moving from A to B
using C >**/
                solveHanoi(N-1, A, B, C);
                /**< move disk remaining from A to C >**/
                move(A,C);
                /**< solve the Hanoi problem of moving N-1 Disks from B to C
using A >**/
                solveHanoi(N-1, B, A, C);
            }
        }

        /**< Move last disk >**/
        // ----------------------------------------------------------------------
        // Move last disk
        // ----------------------------------------------------------------------
        public void move(String A, String B)
        {
            /**< increment step count >**/
            step++;
            /**< Display disk moving  >**/
            con.writeln("Step:"+step+"  Move disk from "+A+" to "+B);
        }
    }
```

Fig. 14. The complete code for the Tower of Hanoi algorithm.

P-Coder also provides a range of other capabilities (not presented here) that can form part of the teaching and learning experience for the novice, including

- construction of UML-styles class diagrams from the P-Coder model;
- object instance creation, inspection, and evaluation;
- import of class sources from other development environments to create skeleton P-Coder models that can be inspected within the P-Coder environment; a skeleton model contains the overall structure of the program (classes, fields, and methods), but not the detailed method code;
- import of library modules (other P-Coder models) that can be copied and pasted (part or whole) into the current P-Coder model;

P-Coder has now been used for three years (2003, 2004 and 2005) to teach within a first computing course in programming and principles. The students were split between two groups; one working towards a Bachelor of Engineering degree (four year), and other group seeking a Bachelor of Technology degree (three year). In both cases the students are totally immersed in the use and application of computer-based technologies. Informally, students have indicated some interest in the approach (few complaints and even some positive words – even from some who thought that this is not how "real" programmers do it).

On a more quantitative basis we have some evidence for the following observations. We are able to compare the results (the total marks obtained for the whole unit, integrated over a number of assessable components including weekly tests, exercises, a major

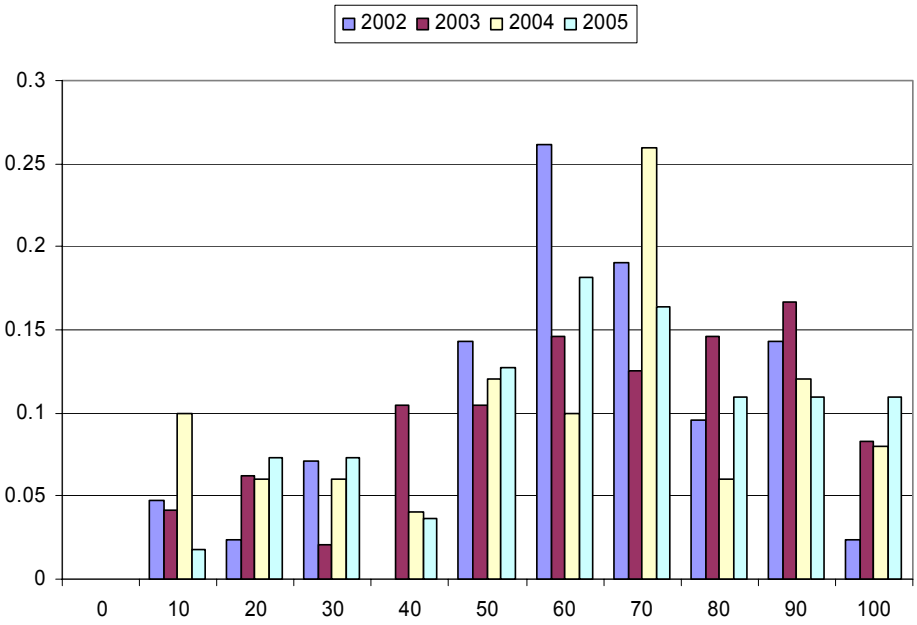

Figure 15: Comparison of 2002 examination results (prior to introduction of P-Coder) and marks for years, 2003, 2004 and 2005

programming project and a written examination) of the 2003, 2004 and 2005 cohorts with the 2002 cohort. For 2002 the basic course was identical, only the tool was changed. Prior to 2003 a more traditional IDE was used for teaching: BlueJ, [Kölling and Rosenberg 2002]. BlueJ has some of the features of P-Coder, but not the pseudocode model – it uses a conventional text editor as the primary specification vehicle.

Figure 15 summarizes and compares the students' results for the four years; the total number of students to complete the unit was 42 (2002), 48 (2003), 50 (2004) and 55 (2005) in each year respectively. While these numbers are relatively small, there is some indication that students at the top end of the mark range are achieving a better result (i.e. higher marks), though there is no real evidence that the average mark has changed significantly. We can be reasonably confident that the performance of the cohorts has not deteriorated. Even though still very preliminary, the results are encouraging.

6. SUMMARY

Explaining algorithms and building correct code are common problems in the computing sciences, and are especially relevant for novice programmers. The P-Coder tool presented here offers a pseudocode model with a strong literate programming basis. It offers the following capabilities:

- a highly visual and readable description of the program and its algorithms;
- the ability to display the program at any level of detail to assist in readability and comprehension;
- the automatic generation of code structure (with some limitations);
- a tight coupling of explanations and documentation (comments) to the specified code segments, all built from a single model of the program.

As a result, the P-Coder model provides a highly readable and internally consistent model for the algorithm/program.

The P-Coder tool that implements these concepts is intended for teaching purposes. It is suspected (but not tested) that it will not scale to very large and complex systems. For more experienced programmers, there were also situations where the tool was valuable in explaining more complex algorithms. Code generated by P-Coder can readily be exported as plain text files to any of the major IDEs. The complete P-Coder models are saved in an XML format.

P-Coder is freely available from http://www.cadplan.com.au for educational purposes.

REFERENCES

CROSS, J.H. 1986. The control structure diagram: An automated graphical stepwise refinement tool with control constructs. PhD thesis, Texas A&M University, College Station, TX.

CROSS, J.H. AND BAROWSKI, L.A. 2002. *The jGrasp Handbook.* School of Engineering, Auburn University.

CROSS, J.H., MAGHSOODLOO, S.H., AND HENDRIX,T.D. 1998. Control structure diagrams: Overview and evaluation. *J. Empirical Software Eng. 3*, 2, 131-158.

CROSS, J.H. AND SHEPPARD, S.V. 1988. Graphical extensions for pseudo-code, PDLs, and source code. In *Proceedings of the ACM 16th Annual Conference on Computer Science* (Atlanta, GA), ACM, New York, 520-528.

DEIMEL, L.E. AND NEVEDA, J.F. 1990. Reading computer programs: Instructor's guide and exercises. CMU/SEI-90-EM-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

ESCALONA, R. 1984. Case study of the methodology of J. D. Warnier to design structured programs as systems documentation. In *Proceedings of the 3rd Annual ACM Conference on Systems Documentation*, ACM, New York, 95-100.

KNUTH, D.E. 1984. Literate programming. *Computer J. 27*, 2, 97-111.

KOLLING, M. AND ROSENBERG, J. 2002. *BlueJ - The Hitch-Hikers Guide to Object Orientation. No 2,* The Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark.

MARTIN, J. AND MCLURE, C. 1985. *Diagramming Techniques for Analysts and Programmers.* Prentice Hall, Englewood Cliffs, NJ.

NASSI, I. AND SHNEIDERMAN, B. 1973. Flowchart techniques for structured programming. *SIGPLAN Notices 8*, 8, 12-26.

ORR, K.T. 1977. *Structured Systems Development.* Yourden Press, New York.

ORR, K.T. 1980. Structured programming in the 1980s. In *Proceedings of the 1980 ACM Annual Conference*. ACM, New York, 323-326.

RAMSEY, N. 2003. Noweb: A simple, extensible tool for literate programming. http://www.eecs.harvard.edu/~nr/noweb.

REYNOLDS,R.G., MALETIC, J.I. AND PORVIN, S.E. 1992. Stepwise refinement and problem solving. *IEEE Software 9*, 5,79-88.

ROBILLARD, P.N. 1986. Schematic pseudocode for program constructs and its computer automation by Schemacode. *Commun. ACM 29*,11, 1072-1089.

SCANLIN, D. 1988. Should short, relatively complex algorithms be taught using both graphical and verbal methods. In *Proceedings of the ACM SIGCSE Conference*, ACM, New York, 185-189.

SHUM, S. AND COOK, C. 2002. Using literate programming to teach good programming practices. http://www.literateprogramming.com/sigcse.pdf.

SPOHRER, J.C., SOLOWAY, E., AND POPPE, E. 1985. Where the bugs are. In *Proceedings of the SIGCHI Conference*, ACM, New York, 47-53.

TAYLOR, R.P., CUNNIFF, N., AND UCHIYAMA, M. 1986. Learning, research, and the graphical representation of programming. In *Proceedings of the Fall Joint Computer Conference*, ACM, New York.

THIMBLEBY, H. 2003. Explaining code for publication. *Software Practice and Experience*,Vol 33, No 10, Aug., pp 975-1001.

VARATEK SOFTWARE INC. 1999. *B-Liner98 Bracket Outliner Users' Guide.* Varatek Software, Andover, MA.

WARNIER, J.D. 1976. *Logical Construction of Programs.* Yourden Press, New York.

Received May 2003; revised September 2005; accepted August 2006