# Teaching design principles in software engineering

**Jocelyn Armarego & Geoffrey G Roy**
*School of Engineering Science*
*Murdoch University, Australia*

The teaching of program design skills to novices is a core problem in software engineering education. This paper discusses the need to develop a good understanding of the fundamental computational principles and identifies some of the key design skills that should be developed by students. The paper proposes that a pseudocode based model has some useful properties in enabling these skills to develop through top down design and through progressive refinement. To demonstrate and test these ideas a pseudocode tool, *P-Coder*, has been developed. This tool provides both graphical and textual elements in an interactive tree structured model. Much of the semantics of a program can be developed graphically before it is necessary to introduce formal programming language syntax. *P-Coder* also provides capabilities to insert code segments, which, when combined with the visual model, enable complete (Java) programs to be created. *P-Coder* is not intended to be a production environment, but rather a tool for developing both knowledge of computational concepts and skill in program design. A preliminary evaluation of student results shows a clear improvement and suggests the approach is worth pursuing.

**Keywords:** programming, program visualisation, program design, algorithm design

## Introduction

Design skills are fundamental to all Engineering disciplines; in fact it is an ability to design that is often used to characterise engineers from other disciplines. In broad term we think of design skills being related to the ability to:

- abstract from specific cases to more general situations
- recognise patterns in both process and product
- apply systematic techniques to problem solving, and
- apply, and adapt, tools and technologies to new problems.

Learning design skills is, however, a non-trivial problem: there are no simple means of teaching them and assessing their development in a totally objective way. Design remains, as perhaps it should, a part of the art that makes an engineer a designer. In this paper we focus on the design of software, and in particular the learning of design skills by novices, in this case students taking their first computing courses in an Engineering school.

The development of software design skills in novices has been a vexing topic for many years. Early interest was sparked in the 1970s (e.g. Dijkstra (1972) and Wirth (1976)) with the advent of a structured approach to programming. This was followed by attempts to devise programming languages to support or complement the structured approach (eg Pascal). Other approaches developed at this time have also contributed to the support of programming tasks and in developing design skills.

More recently the evolution of programming languages (eg C, Visual Basic and Ada) towards O-O principles and the creation of Java are having a major impact on software design. However, although O-O principles have clear advantages when applied to the higher level architectural aspects of program design, they still build on computational principles (sequence, iteration, selection and recursion) that exist in, and are essential to, all programming languages. The understanding of these basic principles is challenging for the novice software designer. The key problem areas can be summarised as:

- difficulties in conceptualising the computational task and its solution, starting from an informal description of the design problem
- confusion between (programming) language syntax and the computational process
- difficulties in devising and understanding the required algorithm

- lack of ability (skill, experience) to understand the flow of computation within a program
- difficulties in using, and appreciating the advantages of, appropriate encapsulation and modularisation concepts, and
- a general lack of understanding of metalevel (e.g. O-O) concepts in programming.

In essence many novice software designers fail to appreciate the big picture while they struggle with the low level syntactical elements of programming languages. In many ways these problems are no different from those in most engineering design areas, with one clear exception. Elsewhere, the fundamental principles are formally taught in introductory courses well before the student is expected to integrate this knowledge into the design task. The principles are also well known and documented through many years of application and evolution. In software design it is often considered that a thorough understanding of principles is not necessary, leading to an unfounded level of confidence in capability that may only become unstuck much later due to design failures (Bergin & McNally, 2000; Buck & Stucki, 2000; Duke, Salzman, Burmeister, Poon, & Murray, 2000; Soloway, 1986).

The goal of this paper is to present a new approach to teaching software design principles and specifically the most basic computational constructs. In particular the intention is to focus on a clear process that enables these constructs to be defined and manipulated before requiring a detailed knowledge of the language implementation. The proposed methodology will also provide a transition from specification to implementation using a process of progressive refinement, with specific support for novice designers.

## 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 made executable. Pseudocode generally includes:

- the use of English like statements (or whatever native language is appropriate) to describe the computational task and/or process
- a small set of reserved words or symbols that are used to describe common processes and actions
- syntactical elements describing the standard computational processes (ie sequence, iteration, selection and recursion), and often
- some graphical notations to add clarity/richness to these descriptions.

Many text based pseudocode variants have been proposed, though these are often informal and not defined with a formal syntax and grammar (ACS, 2002; Adams, 2002; Wells & Kurtis, 1989). Graphical forms of pseudocode have also been proposed, a selection of which are described by Cross and Sheppard (1988). While the goal of each of these is essentially the same – to provide a clear picture of the structure and semantics of the program through either the use of a combination of graphical constructions and/or textual annotations, each style has its strengths and weaknesses in terms of clarity, expressiveness and (most importantly) the overheads involved in using the technique. In addition, as Scanlin (1988) suggests, a combination of text and graphical clues allows pseudocode to make most effective use of all the observer's powers. This is explained by the observation that the textual pseudocode requires processing mainly from the brain's left hemisphere (verbal, logical, sequential) while the graphical elements can effectively utilise the right hemisphere (visual, spatial, simultaneous) at the same time.

## Design languages

With the development of the O-O paradigm, a range of new design techniques has emerged. These provide ways of viewing the underlying computational model within the O-O framework, highlighting or explaining different aspects, for example:

- Use Case Diagrams: show the functional requirements and how the various actors interact with the system
- Class Diagrams: show the static structure and relationships and associations between classes
- State Diagrams: show the permissible states and the transitions that can occur between these states
- Sequence Diagrams: show the temporal dependencies between different actions and objects.

Each view (often based on the Unified Modeling Language (UML) (OMG, 2003)) will have a greater or lesser importance to different stakeholders, different phases of the project and different sections of the project development team.

As shown in Figure 1, the model may reside within an automated environment, often as a CASE (computer aided software engineering) tool supporting UML notation.

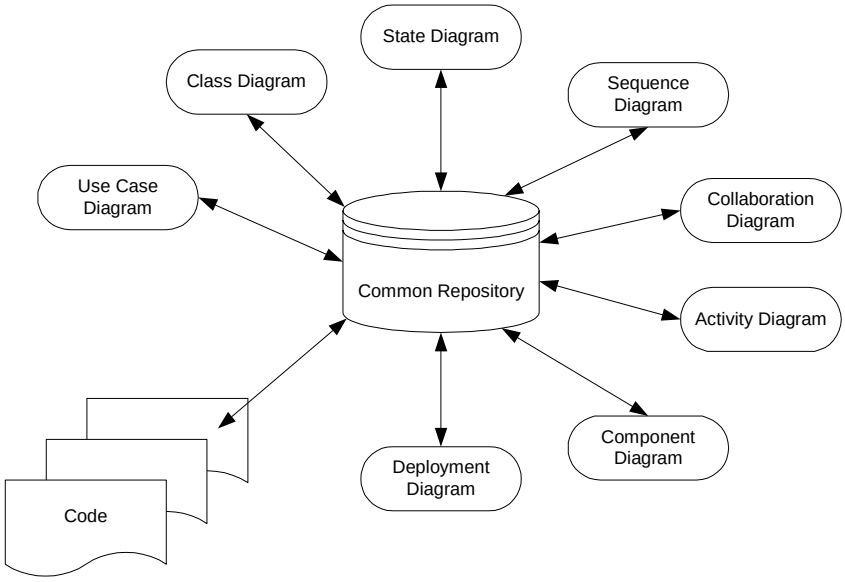



**Figure 1: The UML framework of views**

While UML itself does not require tool support, use of many of its diagramming and modelling elements is facilitated by such support. For example, the repository always contains the common definition of the model and thus ensures consistency between the various views as well as with any code generated from the model. However, CASE tools are generally quite complex to learn and apply effectively to real software tasks: they tend to be used on large projects where the payoffs can justify the overheads, and therefore are typically less suitable for novices.

For novices our teaching focus is at a more micro level, constructing basic computational elements. It is our experience that many of the problems for novices originate here, rather than with the more abstract O-O principles or higher level design concepts usually captured in a UML described model. There are, however, some elements of UML that can, and should, be included in novice teaching as they facilitate a basic understanding of software design, even where the design task is quite simple.

## Design principles

Tools developed to assist students in understanding at this micro level should address the following design principles:

- *Literate programming* (Knuth, 1984; Shum & Cook, 2002) provides a perspective on the programming process where the focus is on integrating (formally) the informal descriptions of the program's functions with the formality of the programming language instructions. While the inclusion of comments in code is generally accepted as good practice there are few tools that require, or support, it in a tightly coupled and formal way. There is some evidence ((Bonar & Soloway, 1983; Bruckman & Edwards, 1999)) to indicate that a significant proportion of errors in novice programming are caused by a confusion of natural language semantics with the more limited (but precise) semantics of the programming language. As a result there may be some value in leveraging the use of natural language expressions to better support the programming process

- *Stepwise refinement***.** The task of creating a program from a specification is an open problem: of the many concepts, ideas, strategies and processes that can be proposed none provide all the answers. However, stepwise, or progressive, refinement is one strategy that appears often (Reynolds, Maletic, & Porvin, 1992; Wirth, 1971). Quoting from Reynolds et al (1992; p 80):

  > …stepwise refinement can be viewed as a sequence of elaborations that result in the formation of a program in a target language from an initial function specification

  In some ways this principle is still at the core of teaching novice designers, and central to the goals for the use and development of pseudocode based technologies/tools. Novices struggle with understanding how a computation should be performed - the problem needs to be approached gradually and progressively
- *Encapsulation and information hiding*. Encapsulation is concerned with the containment of code within program elements that can stand alone and integrate with other program elements through well defined interfaces (Blair, Gallagher, Hutchison, & Sheperd, 1991; Booch, 1991; Wirfs-Brock, Wilkerton, & Weiner, 1990). This facilitates separate compilation and the effective management of large and complex systems. Information hiding is aimed at ensuring that attributes and operations are defined so that:
  - their visibility is limited to those parts of the system where they are needed
  - the ability to observe and/or modify data or behaviour is restricted to those parts of the system where the designer explicitly allows such capabilities.

  These principles are integral to the effective re-use of proven and well tested components in the large software libraries that form essential tools for the software designer. Most engineering design tasks rely on standard components: software should be no different
- *Modularisation*. Most complex design tasks are solved by breaking them down into manageable parts. These modules must be of a size or complexity so their functional requirements can be clearly stated and implemented without being too concerned about the operation of the whole system (Yourdon & Constantine, 1979). This requires a clear definition of how the module interfaces with other parts of the system and a description of any conditions that can be applied to ensure the integrity of the module. Developing skills in modular design is essential for teaching novices. “Rules of thumb” like limiting the complexity of an operation to a single task can assist in knowing when further modularisation is required
- *Model representations*. The design process for software, like other areas of engineering, requires the designer to have an appropriate mental model. For software this is commonly taken as the textual representation. In reality this is just one view of the design solution, and often not the most effective. Hence the emergence of modelling languages such as UML to describe the complex behaviour of software. The idea that an underlying model can have several views to explain different elements of the semantics of the program is central to the development of a good appreciation of the role of the design model and should be introduced at an early stage of teaching.

Regardless of what programming paradigm is adopted, early progress and ultimate success for the novice will be closely related to how well the fundamentals of computational primitives (ie sequence, iteration, selection and recursion) are understood. Our approach has been to develop a model for software design that focuses on these basic elements, while introducing a limited subset of O-O concepts. The goal is to provide a framework that can be extended as the student’s understanding and design sophistication grows.

## A modelling framework

In simple terms we can consider many aspects of a computer program to be represented by an abstract tree. A tree model has a single root node, then branches (one or more) that in turn branch until they terminate at leaves. Abstract trees have a number of important properties for design. In particular they:

- provide for multiple levels of abstraction so that details at lower levels can still be defined, hidden, or expanded as required
- provide a natural hierarchy that can model many computational concepts
- facilitate top down design, and
- allow for, and encourage, progressive refinement of the design.

While these properties are not sufficient to solve a complex design problem entirely, they can contribute to effective design processes. Pseudocode is generally presented in this way, though often informally. Simple indentation rules, for example, describe a tree structure. It most naturally follows a top down approach (describe the major operations first) and it encourages (requires) a progressive refinement by expanding each step into sub-steps of increasing detail.

A pseudocode model of a program must be able to represent the required algorithm using some combination of the computational primitives (sequence, selection, iteration and recursion). It may also require additional constructs to express:

- design framework concepts that reflect encapsulation and modularisation (data, functions, methods, classes, packages)
- scope concepts that facilitate data/process visibility schemes (class and local data concepts, get/set access controls, etc)
- relationship concepts that describe dependencies between program components (extends, uses, etc)
- temporal elements of the required computation (sequence and concurrency)
- exception handling, pre and post conditions
- macro level architectures through distributed or networked capabilities
- data storage, transfer and I/O requirements
- some target language specific elements.

For even the simplest of programs some of these constructs are required, so they cannot be ignored entirely. Not all need be present in a single tool: some may be application domain specific. It is most probable that some syntactical elements will have to be introduced to achieve this - natural language by itself will not be adequate. Simple tree models, assumed to underlay the pseudocode, will not accommodate all these concepts directly: some extensions will be required.

## Pseudocode tools

To be effective, the use of pseudocode must be supported by an appropriate tool. Without some tool support there is a tedious level of re-writing, additions and deletions to allow the pseudocode description to grow and evolve effectively. A number of these have been proposed, and each of the following examples has an abstract tree model as its foundation:

- *B-liner* (Varatek Software Inc., 1999) is based on the Warnier-Orr diagramming model (Escalona, 1984; Orr, 1980; Warnier, 1976) where concepts/relations are organised into a hierarchical tree using a 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. Algorithm development proceeds top down, adding the detail as the refinement proceeds. A “bracket” contains all the children 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/expand each branch of the tree to aid its readability as the size of the tree grows. Details are added by expanding each node with a new bracket to its right. *B-liner* provides a tool for describing algorithms at varying levels of detail. It supports step wise refinement, but there is no way of specifying, or generating, executable code in the model. It is primarily a tool for algorithm specification
- *SchemaCode* (Robillard, 1986) is an older tool, but more specifically aimed at the task of programming. It is targeted more directly at the generation of executable code and relies on the user inserting some (all) program structures to match the target language, as well as the full syntax of the required computations. Starting from the top (the root of the tree), there is an implied sequence of nodes as we move down the diagram. Branches are added to define/refine computational steps, and opened/closed to explore their contents. The program can be developed incrementally by stepwise refinement, with the leaf nodes (ultimately) containing actual code. Intermediate nodes contain the text notations, increasing the readability of the diagram that otherwise appears as the actual code of the program
- *jGrasp* (Cross & Barowski, 2002) is not really a pseudocode tool, though it does offer some useful contributions. In *jGrasp*, the starting point is the actual (syntactically correct) code. The tool is based on the concept of Control Structure Diagrams (CSD) (Cross, Maghsoodloo, & Hendrix, 1998; Cross

& Sheppard, 1988). The CSD diagram is created from the code and is displayed along with the code, to provide a more readable display of the structure of the program (and the underlying algorithm) after the code is written. Various graphical annotations are added to the code to clarify the computational processes by assisting the readability of the program. The model is changed by directly editing the code: the CSD changes are then available to view. *jGrasp* allows the CSD to be rolled up and out to hide and display tree branches are varying levels of detail. It provides a "full" development environment to support programming, and includes a range of capabilities for editing, compiling, debugging and execution of program as well as generating UML styled class diagrams.

While each of these tools provides some level of support for program design they each have their shortcomings: *B-Liner* does not support a progressive refinement process leading towards code development; *jGrasp* provides no support at the design stage – but it does assist in explaining the code; *SchemaCode* has many of the required elements for an effective tool and supports a range of programming languages (not Java yet), but is perhaps a little dated in its presentation and user interface design.

## A new pseudocode tool: *P-Coder*

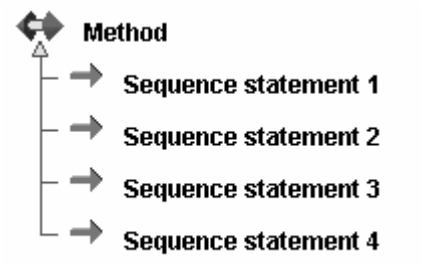
*P-Coder* has been developed in an attempt to meet the design goals described earlier. It has the following general capabilities for representing a model of a program using a pseudocode notation:

- the model is tree structured, with the root of the tree being at the top left
- the tree is composed of nodes, each providing a specific semantic contribution to the model
- primitive nodes are provided to represent sequence, iteration, selection and recursion
- higher level nodes provide additional programming and O-O constructs like packages, classes, methods, class fields, and exception handling
- the tree model is built interactively and progressively using stepwise refinement with full editing capabilities, including the ability to cut and paste tree components (nodes and sub-trees)
- model building can take place in both the tree view of the model and the class diagram view. This allows design to proceed within the pseudocode model, as well as at the class level, as elements of O-O are introduced into the teaching programme
- code skeletons can be automatically generated from the model in the chosen language
- each tree node may also contain a set of detailed parameters/code segments that can be used to build a complete executable program. A program can be completed in *P-Coder* or exported to another programming environment
- It is currently Java focussed, though the concepts could be extended to other languages.

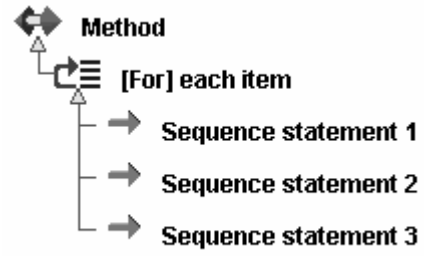
While in some ways *P-Coder* is quite similar to a number of the already proposed graphical pseudocode tools, it uses a "richer" set of icons to show the semantics of the nodes and a more flexible set of graphical manipulation capabilities. The *P-Coder* model contains all the essential information to define much of the code for the program without the user needing to know very much about the syntax of the target language.

The *P-Coder* expressions for the four basic computational elements are shown in Figure 2. While it might seem trivial to examine the computational processes at this level, this is where many students have the greatest difficulty. This problem is then exacerbated when we introduce language specific syntax that can sometimes cloud the clarity of the computational steps.

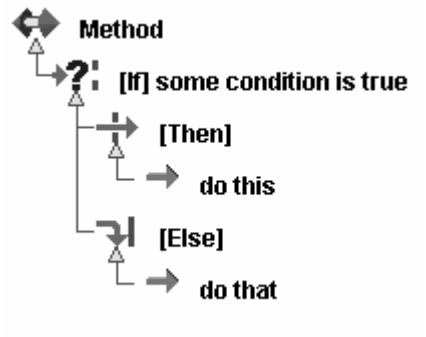
Figure 2(e) shows an example *P-Coder* model showing the solution to the quadratic equation $ax^2+bx+c=0$. The icons used for each node provide an immediate visual clue to the semantics, and the notes (placed to the right of each node) provide some additional information to clearly describe the required operation/semantics. There are specific node types to describe the computational primitives, Figure 2(a) – (d), plus others to define higher level structures such as packages and classes and specialised nodes provided as a part of imposing some stylistic guidelines (eg keep your data declarations together at the top of the class or method).



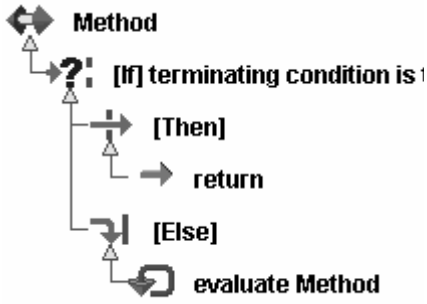

(a) Sequence



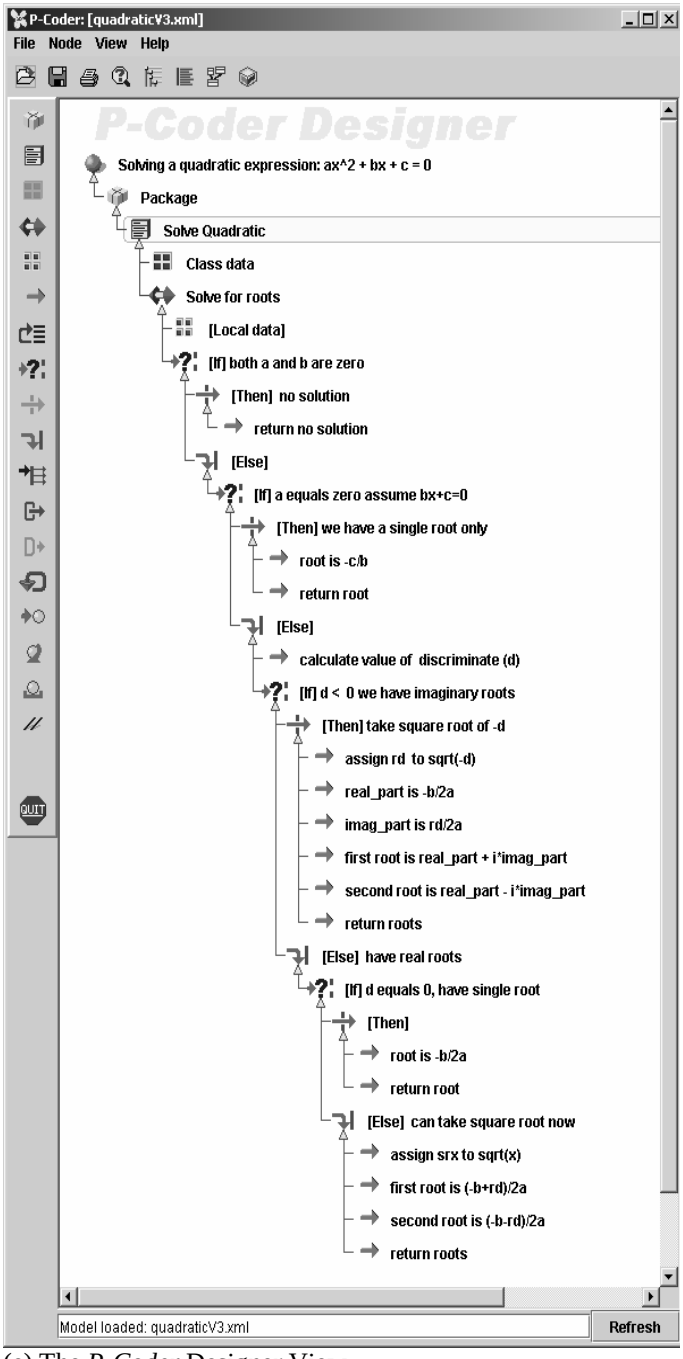

(b) Iteration




(c) Selection




(d) Recursion




(e) The *P-Coder* Designer View

**Figure 2: The *P-Coder* computational primitives (a, b, c, d) and designer view (e)**

In an O-O framework attributes and operations are collected into containers (the Class) that describe some properties and behaviour. The Class View in *P-Coder* provides this view of the model, to show class relationships. Figure 3 shows the generalisation/specialisation relationships ("extends"), but others ("uses" - association or aggregation/ composition and "interface" - classes with no implementation) are also supported.

While the Designer View does not provide sufficient detail to show all the properties of the classes, especially the relationships between them, the Class View (derived from the UML specification, but only supporting a part of the notation) does. Here the class properties are shown in a summary form and the class associations shown to clearly indicate super and sub-classes. The Class View is generated from the

*P-Coder* model, though the user controls its layout. The user can manipulate both the Designer and Class views of the model interactively. Classes can be added and edited, class attributes can be added and edited, operations (methods) can also be added and edited. As these two views are just different perspectives of the same model, editing one causes changes in both views.

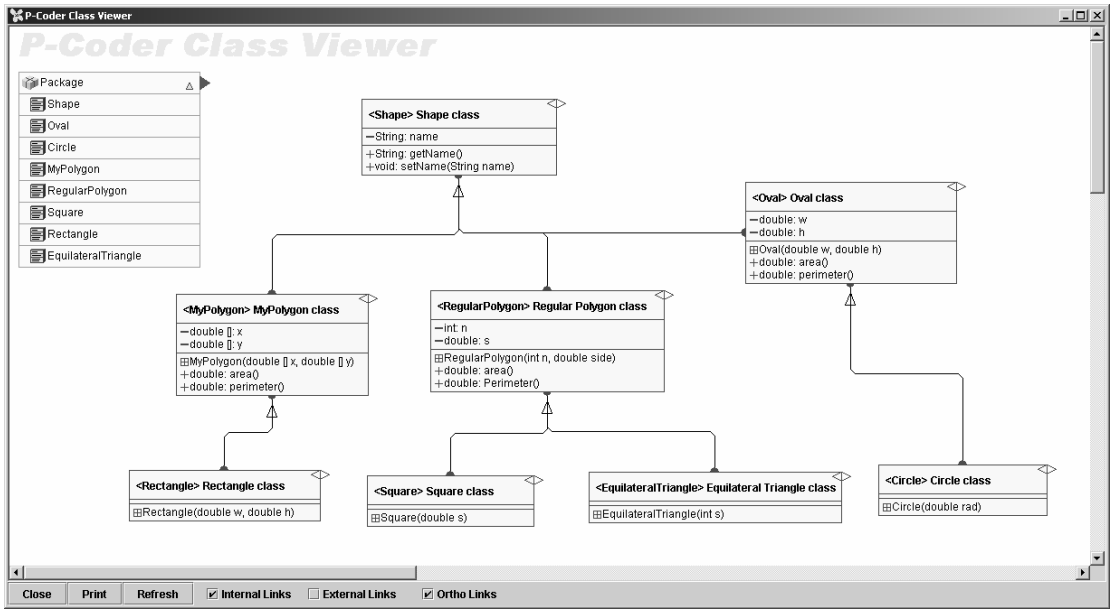



**Figure 3: The class diagram for the *Shapes* model**

*P-Coder* is used to teach the very basics of computation. The Designer View allows users to begin their thinking in terms of primitive computational processes, with descriptions of the tasks and the proposed steps necessary to solve the problem. The tool allows these initial statements to be modified and refined by extending and adding more detail into the model. In this way the basic computational steps are described and refined.

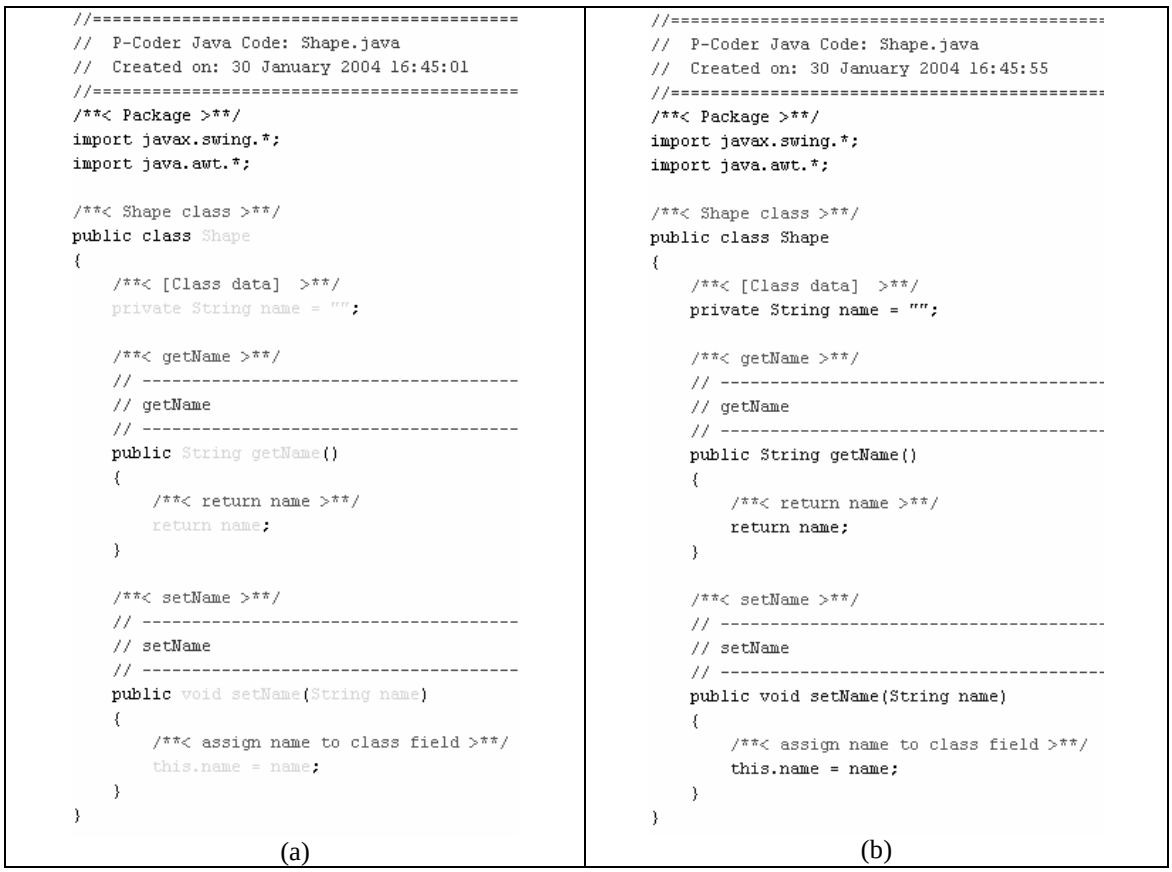
Once the basic program elements are in place (computational steps and their organisation into classes) much of the actual code for the program can be generated automatically as shown in Figure 4(a). Here the code segments that can be automatically generated from the model (with some default settings) are shown while those that remain to be added are faded out. The user thus can be shown a lot about how the final program should look without being too concerned (at this stage) with the syntax details of the chosen language (Java here). In many ways the design of the program is now complete, though more detail is required to fully operationalise it. The task for the user is to add the missing code segments and to complete the code as shown in Figure 4(b). This is done using a set of details dialogs (one for each node type in the Designer and Class Views) as shown, for example, in Figure 5. Because we know what information is required for the given node, we can prompt for the essential items. While the user must know something about the target programming language at this stage, we can save a lot of effort in learning a much larger subset of language elements before being able to write the simplest of programs.

The final code also contains the pseudocode (as comments) embedded within it and tightly coupled to the actual executable code that has been derived and added as shown in Figure 4. This makes the resulting program ‘literate’ in the sense that it contains much of the essential explanations and documentation.

*P-Coder* is not intended for large scale programming tasks. These are more effectively handled by one of the many interactive development environments. It is, however, perfectly adequate for most teaching situations.

## Preliminary results of using *P-Coder*

*P-Coder* is currently under evaluation within our School: it is being used in the first and second Engineering Computing courses. These provide an introduction to computing methods and programming and include a range of topics in discrete mathematics using the Java language as a vehicle to teach both the programming and mathematics concepts. We can report a comparison of final grades for these courses comparing the 2002 (prior to the introduction of *P-Coder*) and the 2003 results. For both years the curriculum was essentially the same and was delivered by the same teaching staff. These results are very preliminary and a more detailed examination of the students' actual design processes is currently underway.



```
//============================================
//  P-Coder Java Code: Shape.java
//  Created on: 30 January 2004 16:45:01
//============================================
/**< Package >**/
import javax.swing.*;
import java.awt.*;

/**< Shape class >**/
public class Shape
{
    /**< [Class data]  >**/
    private String name = "";

    /**< getName >**/
    // ----------------------------------------
    // getName
    // ----------------------------------------
    public String getName()
    {
        /**< return name >**/
        return name;
    }

    /**< setName >**/
    // ----------------------------------------
    // setName
    // ----------------------------------------
    public void setName(String name)
    {
        /**< assign name to class field >**/
        this.name = name;
    }
}
```

(a)

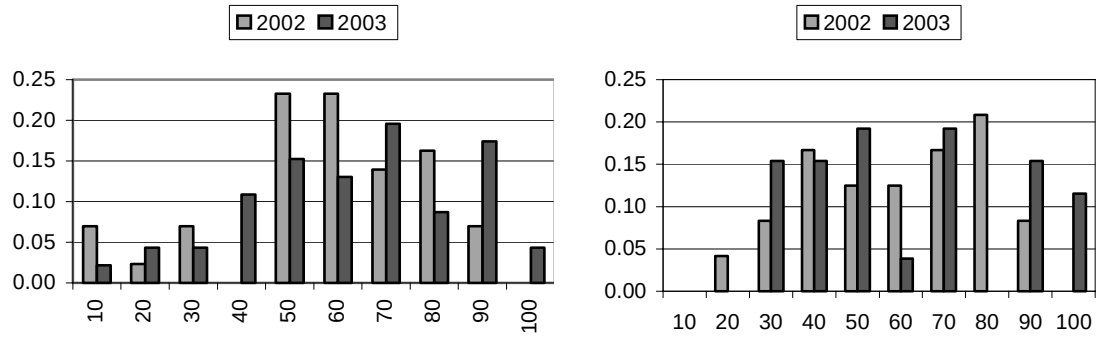
```
//============================================
//  P-Coder Java Code: Shape.java
//  Created on: 30 January 2004 16:45:55
//============================================
/**< Package >**/
import javax.swing.*;
import java.awt.*;

/**< Shape class >**/
public class Shape
{
    /**< [Class data]  >**/
    private String name = "";

    /**< getName >**/
    // ----------------------------------------
    // getName
    // ----------------------------------------
    public String getName()
    {
        /**< return name >**/
        return name;
    }

    /**< setName >**/
    // ----------------------------------------
    // setName
    // ----------------------------------------
    public void setName(String name)
    {
        /**< assign name to class field >**/
        this.name = name;
    }
}
```

(b)

**Figure 4: The *P-Coder* code view**


**Figure 5: A typical Details Dialog to enter code segments**

*P-Coder* is introduced and used almost exclusively throughout the first course. In the second course a transition is made from *P-Coder* to a traditional programming environment (*BlueJ* (Kölling, 2002)) which, although it provides some quite innovative design/programming tools, is still essentially text based. The results (see Figure 6) indicate a clear improvement after the introduction of *P-Coder*. The first course was taken by about 40 students, and the second course by about 25 students, who are close to (but not precisely) a subset of the first group. The statistical significance of the change is not as clear, but the results look promising enough to give us the confidence to continue working with this new software design environment. Anecdotally, teaching staff in a follow on programming unit has commented that there were no "weak" programmers in the cohort.




(a) First course

(b) Second course

**Figure 6: Comparisons of pre and post *P-Coder* results**

## Summary

Pseudocode, as a concept for assisting with the developing of programming and program design skills to novices, has been around for some time. While it is often suggested as a useful strategy for improving the teaching and learning processes for novice programmers, its general acceptance has been limited.
The value of pseudocode has probably been restricted by the lack of good tools to facilitate its use while providing the novice with an interesting environment to work with. This paper has presented a new support tool that might go some way to redress this problem.
However, there are still some important research problems relating to the use of pseudocode concepts, for example:

- Can we identify measurable effects in improved design outcomes where a well structured pseudocode approach is adopted and enforced?
- At what point do we stop the refinement process and require the programmer to develop the code in the target language?
- Will a tool (like *P-Coder*) maintain the interest of novices for long enough so that the understanding of the basic computational processes is well developed and the design habits well established?
- How do we integrate the full range of O-O principles, and how should the balance between these and the procedural elements be maintained?

These are still open questions that we are pursuing with further research into the learning of programming design skills by novices.

## References

ACS. (2002). *Appendix to Programming and Software Technology*. Retrieved from http://www.acs.org.au/static/training/exam4.htm

Adams, M. (2002). *Pseudocode to C++*. Retrieved from http://sky.fit.qut.edu.au/~adamsmj

Bergin, J., & McNally, M. (2000). *Non-Programming Resources for an Introduction to CS: A collection of resources for the first courses in Computer Science.* Paper presented at the ITiCSE 2000.

Blair, G., Gallagher, J., Hutchison, D., & Sheperd, D. (1991). *Object-Oriented Languages, Systems and Applications*: Halsted Press, New York, NY.
Bonar, J., & Soloway, E. (1983). *Uncovering Principles of Novice Programming.* Paper presented at the Proceedings 10th SIGPLAN Symposium on Principles of Programming Languages
Booch, G. (1991). *Object-Oriented Design With Applications*: Benjamin/Cummings, Meno Park, CA.
Bruckman, A., & Edwards, E. (1999). *Should We Leverage Natural-Language Knowledge An Analysis of user Errors in an Natural-Language-Style Programming Language.* Paper presented at the Proc SIGCHI
Buck, D., & Stucki, D. J. (2000). *Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development.* Paper presented at the SIGCSE 2000, Austin
Cross, J. H., & Barowski, L. A. (2002). The jGrasp Handbook: School of Engineering, Auburn University.
Cross, J. H., Maghsoodloo, S. H., & Hendrix, T. D. (1998). Control Structure Diagrams: Overview and Evaluation. *Journal of Empirical Software Engineering, 3*(2), 131-158
Cross, J. H., & Sheppard, S. V. (1988). *Graphical Extensions For Pseudo-Code, PDLs, and Source Code.* Paper presented at the ACM 16th Annual Conference on Computer Sceince, Atlanta
Dijkstra, E. W. (1972). *Structured Programming*: Academic Press.
Duke, R., Salzman, E., Burmeister, J., Poon, J., & Murray, L. (2000). *Teaching Programming to Beginners - choosing the language is just the first step.* Paper presented at the Proc Australian Conference on Computer Science Education, Melbourne
Escalona, R. (1984). *Case Study of the Methodology of J. D. Warnier to Design Structured Programs as Systems Documentation.* Paper presented at the Proceedings 3rd Annual ACM Conference on Systems Documentation
Knuth, D. E. (1984). Literate Programming. *The Computer Journal, 27*(2), 97-111
Kölling, M. (2002). *The BueJ Tutorial, vers 1.4*: Maersk Institute, University of Southern Denmark
OMG. (2003). *Unified Modelling Language Specification* (1.5 ed.): Object Managment Group, Needham MA.
Orr, K. T. (1980). *Structured Programming in the 1980s.* Paper presented at the Proceedings 1980 Annual Conference
Reynolds, R. G., Maletic, J. I., & 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. *Communications of the ACM, 29*(11), 1072-1089
Shum, S., & Cook, C. (2002). *Using Literate Programming to Teach Good Programming Practices*. Retrieved from http://www.literateprogramming.com/sigcse.pdf
Soloway, E. (1986). Leaning to Program = Leaning to Constrcut Mechanisms and Explanations. *Commumications of the ACM, 29*(9), 850-858
Varatek Software Inc. (1999). *B-Liner98 Bracket Outliner Users' Guide*. Andover, MA: Varatek Software.
Warnier, J. D. (1976). *Logical Construction of Programs*. NY: Yourden Press.
Wells, M. B., & Kurtis, N. L. (1989). *SIGSE 89*
Wirfs-Brock, R., Wilkerton, B., & Weiner, L. (1990). *Designing Object-Oriented Software*: Prentice-Hall, Englewood Cliffs, NJ.
Wirth, N. (1971). Program Development by Stepwise Refinement. *Communications of the ACM*, 221-227
Wirth, N. (1976). *Algorithm + Data Structures = Programs*: Prentice Hall.
Yourdon, E., & Constantine, L. L. (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.*: Prentice-Hall, Englewood Cliffs, NJ.

**Please cite as:** Armarego, J. & Roy, G. (2004). Teaching design principles in software engineering. In R. Atkinson, C. McBeath, D. Jonas-Dwyer & R. Phillips (Eds), *Beyond the comfort zone: Proceedings of the 21st ASCILITE Conference* (pp. 67-77). Perth, 5-8 December. http://www.ascilite.org.au/conferences/perth04/procs/armarego.html